

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

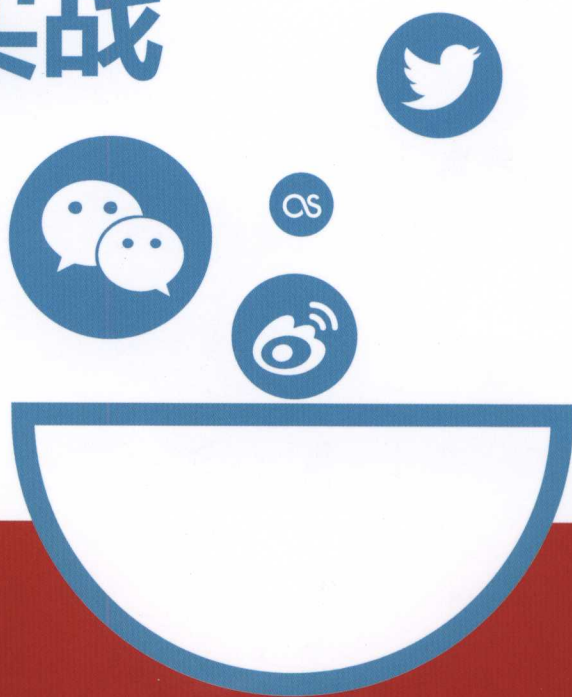
详解React Native应用从创建、开发到发布的全过程，展示各组件和API的用法

实战为王，通过典型项目案例，让读者快速掌握React Native应用开发

React Native

移动开发实战

袁林◎编著



- 书中所有内容都配合详细的实例和源代码进行讲解
- 全面涵盖React Native组件、API、布局、第三方组件及原生接口开发等内容
- 详解React Native的开发工具、命令行工具及各种调试工具的使用
- 详细讲解一个电商App项目案例的开发过程，提高读者的实战开发水平
- 涉及软件开发流程、应用架构设计、代码重构，以及原生平台与跨平台开发等



机械工业出版社
China Machine Press

内容简介

本书以实战开发为主旨，以React Native应用开发为主线，以iOS和Android双平台开发为副线，通过完整的电商类App项目案例，详细地介绍了React Native应用开发所涉及的知识，让读者全面、深入、透彻地理解React Native的主流开发方法，从而提升实战开发水平和项目开发能力。

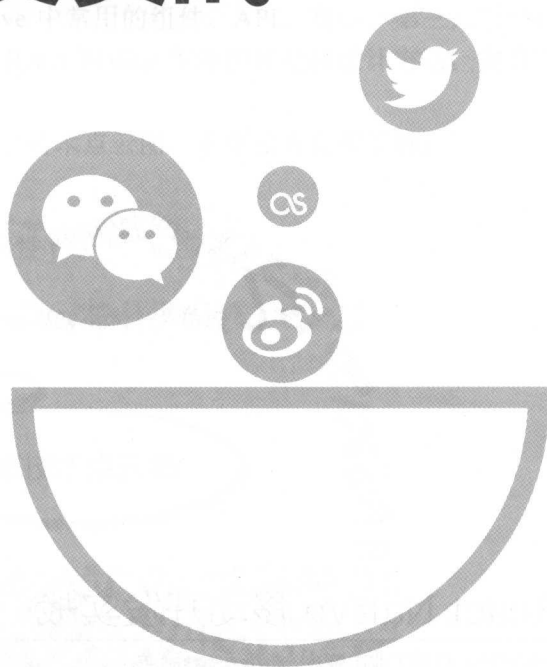
本书共12章，分为4篇，涵盖的主要内容有搭建开发环境、Nuclide、各种命令行工具（Git、Node.js）、布局与调试、组件、API、第三方组件、基于Node.js的服务器、fetch API、AsyncStorage/SQLite/Realm数据库存储、原生平台接口开发、redux开发框架、应用打包与发布、热更新与CodePush等。

本书适合iOS和Android原生平台应用开发者，以及有兴趣加入移动平台开发的JavaScript开发者阅读。当然，本书也适合相关院校和社会培训学校作为移动开发的教材使用。

React Native

移动开发实战

袁林◎编著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

React Native 移动开发实战/袁林编著. —北京: 机械工业出版社, 2017.6

ISBN 978-7-111-57179-7

I. ①R… II. ①袁… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字 (2017) 第 146546 号

React Native 移动开发实战

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 欧振旭

责任校对: 姚志娟

印 刷: 中国电影出版社印刷厂

版 次: 2017 年 7 月第 1 版第 1 次印刷

开 本: 185mm×260mm 1/16

印 张: 20

书 号: ISBN 978-7-111-57179-7

定 价: 69.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光/邹晓东

前言

随着手机和移动互联网技术的日益成熟，移动应用的领域也从如何开发，发展到如何更高效、更低成本地开发。传统的原生平台（iOS、Android）开发技术虽然比较成熟，但由于开发效率和成本的限制，已经越来越无法满足移动互联网应用的开发需求。

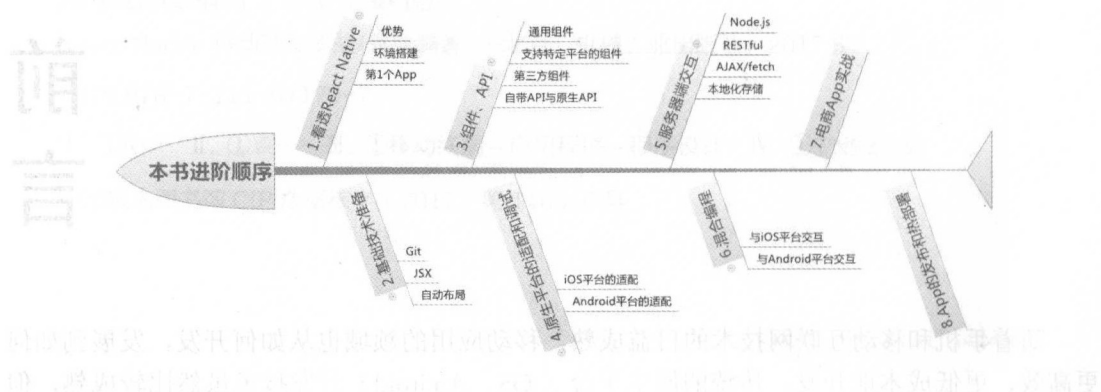
所以，具有简单、迅速、跨平台的优势，而且基于 Web 开发语言和布局技术的 React Native 得以迅速流行，并一举夺得跨平台开发技术的头筹。

目前市场上大多数 React Native 书籍主要以翻译和讲解官方文档为主，并未从开发实际应用出发，通过典型案例来指导读者提高开发水平。本书以实战为主旨，通过完整的电商类 App 项目实例，来介绍 React Native 中常用的组件、API、布局、第三方组件和原生接口，让读者全面、深入、透彻地理解 React Native 主流的开发和设计方法，提升实际开发水平和项目实战能力。

本书涉及的概念较多，下面给出一个技术点云图，希望读者有所了解。



本书的进阶顺序，也给出如下一个图，便于读者了解。



本书特色

1. 每一步都有详细的源码和实例参考

为了便于读者理解本书内容，提高学习效率，本书的所有内容都有详细的源码和实例参考。对于这些源码和实例，作者均亲自编写和验证，杜绝复制、粘贴代码以敷衍读者的不负责任行为。本书源码可以在 <https://coding.net/u/learnreactnative/p/learnreactnative-sourcecode/git> 里下载。

2. 内容涵盖React Native开发的各个方面

本书涵盖 React Native 组件、API、布局、第三方组件以及原生接口等 React Native 应用开发的各个方面，尽量保证不出现知识“死角”。凡是涉及的一些技术（如原生、瀑布流、耦合性和 JSON），也给出了概念或原理解释。

3. 结合工具助力更高效的React Native开发

在本书“实战”讲解的过程中，详细介绍了 React Native 开发工具 Nuclide 的使用、React Native 命令行工具的用法及各种调试工具（包括布局、断点及实时加载等）的使用，不仅教读者如何开发，还教读者如何更高效地开发。

4. 项目案例典型，实战性强，有较高的应用价值

本书以开发一个电商类应用为例，涵盖了 React Native 应用开发中会用到的所有重点知识，设计和源码做到拿来可用，方便应用开发者随时查阅和参考。

5. 收获的不仅仅是React Native平台和编码

对于一些学习能力较强的读者，完全可以在 React Native 开发文档的帮助下快速学习和掌握 React Native。而本书希望读者在掌握平台和编码之外，还能够了解实际应用开发过程中涉及的软件开发流程、应用架构设计、代码重构技巧，以及原生平台与其他跨平台开发的相关知识，让读者融会贯通地理解应用开发技术。

本书内容及知识体系

第1篇 React Native入门和基础（第1~2章）

本篇介绍了跨平台开发的主流方案和 React Native 基础知识，主要包括开发环境搭建、

React Native 命令行工具和 React Native 布局调试。

第2篇 React Native应用开发实战（第3~7章）

本篇介绍了 React Native 实际应用开发中常用的技术，主要包括基本组件、使用第三方组件、搭建基于 Node.js 的服务器为应用绑定真实数据、fetch API、AsyncStorage/SQLite/Realm 数据库存储、更多 React Native 组件和 API 的用法、原生平台接口开发等。

第3篇 React Native混合编程（第8~10章）

本篇主要总结和回顾了前 7 章所开发的电商类应用的技术和架构，主要包括应用的文件结构、Flexbox 的整体布局、应用的逻辑结构、应用的通信过程及进一步改进的地方和思路，其中就包括了 redux 开发框架。

第4篇 App的发布和更新（第11~12章）

本篇主要介绍了 React Native 应用打包和发布的全过程，配以详细的截图说明，并且对 React Native 应用发布后的热更新实现和方案 CodePush 做了详细的示例说明。

适合阅读本书的读者

- React Native 学习人员；
- iOS 平台应用开发工程师；
- Android 平台应用开发工程师；
- Web 前端开发工程师；
- Node.js 服务端开发工程师；
- 计算机相关专业的学生；
- 专业培训机构的学员；
- 软件开发项目经理。

本书作者

本书由袁林主笔编写。其他参与编写的人员有高旭、贺庆端、黎林、李伟浩、刘成智、刘利、刘源、谭建利、吴贵文、吴娟、夏秀英、肖太平、郑星。

致谢

感谢本书的编辑，让我有机会和本书结缘。感谢我的伙伴们：邵长磊、刘冬冬、袁方、袁满、翟绍虎、洪敏、郭晨光及张砚，与我一起探讨新技术并和 React Native 结缘。感谢我的妻子韩丽、女儿可可及我的父母，写作占用了我很陪伴家人的时间和精力，正是有了家人的支持，才得以坚持下去。

最后还要感谢读者，本书的价值因你们而存在。

编者

目录

前言

第 1 篇 React Native 入门和基础

| | |
|---|----|
| 第 1 章 为什么要学习 React Native | 2 |
| 1.1 看透 React Native | 2 |
| 1.1.1 React Native 与 React.js | 2 |
| 1.1.2 React Native 的跨平台 | 3 |
| 1.1.3 解剖 React Native 应用的结构 | 4 |
| 1.2 React Native 的特点 | 5 |
| 1.2.1 其一：Learn Once, Write Anywhere | 5 |
| 1.2.2 其二：简单易学的开发语言 | 6 |
| 1.2.3 其三：接近原生应用的性能和体验 | 7 |
| 1.2.4 其四：完善的生态系统 | 7 |
| 1.3 搭建 React Native 开发环境 | 9 |
| 1.3.1 安装原生开发工具——Android | 9 |
| 1.3.2 安装原生开发工具——iOS | 11 |
| 1.3.3 安装 Node.js | 12 |
| 1.3.4 安装 React Native | 13 |
| 1.3.5 安装其他辅助工具 | 14 |
| 1.4 第一个 React Native 应用 | 16 |
| 1.4.1 初始化项目 | 16 |
| 1.4.2 运行项目 | 17 |
| 1.4.3 调试项目 | 18 |
| 1.5 小试牛刀——更改 React Native 项目源码 | 18 |
| 1.6 小结 | 20 |
| 第 2 章 全局解析 React Native 开发的基础技术 | 21 |
| 2.1 开发具备的基础知识说明 | 21 |
| 2.2 Git 版本控制工具 | 22 |
| 2.2.1 安装 Git | 22 |
| 2.2.2 Git 常用命令 | 22 |
| 2.3 React Native 的 JSX 解决方案 | 24 |
| 2.4 React Native 的 Flexbox 布局 | 25 |
| 2.4.1 flexDirection 设置组件的排列 | 26 |

| | | |
|-------|-------------------------|----|
| 2.4.2 | flexWrap 设置是否换行 | 28 |
| 2.4.3 | justifyContent 设置横向排列位置 | 30 |
| 2.4.4 | alignItems 设置纵向排列位置 | 31 |
| 2.4.5 | alignSelf 设置特定组件的排列 | 33 |
| 2.4.6 | flex 设置组件所占空间 | 34 |
| 2.5 | 如何调试 React Native 项目 | 35 |
| 2.6 | 实战——设计一个电商 App | 37 |
| 2.6.1 | 电商 App 的模块划分 | 37 |
| 2.6.2 | 设计首页布局 | 41 |
| 2.6.3 | 实现搜索栏 | 44 |
| 2.6.4 | 设计轮播广告 | 46 |
| 2.6.5 | 展示商品列表 | 51 |
| 2.6.6 | 实现交互功能和状态栏 | 52 |
| 2.7 | 小结 | 56 |

第 2 篇 React Native 应用开发实战

| | | |
|-------|------------------------------------|----|
| 第 3 章 | React Native 的组件 (1) | 58 |
| 3.1 | 创建新的电商 App | 58 |
| 3.1.1 | 移植旧电商项目 | 58 |
| 3.1.2 | 重构现有的代码 | 60 |
| 3.2 | 完善搜索框功能——TextInput 组件 | 64 |
| 3.2.1 | 搜索提示框 | 64 |
| 3.2.2 | 调试搜索结果 | 66 |
| 3.2.3 | 优化搜索框样式 | 67 |
| 3.3 | 完善轮播广告——Image 组件 | 68 |
| 3.3.1 | 使用网络图片 | 68 |
| 3.3.2 | 使用本地图片 | 69 |
| 3.3.3 | 添加指示器组件 | 71 |
| 3.4 | 完善商品列表——ListView 组件 | 73 |
| 3.4.1 | 对图片资源进行重构 | 74 |
| 3.4.2 | 重新定义商品模型 | 75 |
| 3.4.3 | 商品布局的优化 | 76 |
| 3.5 | 拖曳刷新列表——RefreshControl 组件 | 80 |
| 3.6 | 添加页面跳转功能——Navigator 组件 | 83 |
| 3.7 | 二级页面的跳转——TouchableOpacity 组件 | 86 |
| 3.8 | 实现页面间的数据传递 | 89 |
| 3.9 | 小结 | 90 |
| 第 4 章 | React Native 的组件 (2) | 91 |
| 4.1 | 只支持特定平台的组件 | 91 |
| 4.1.1 | 实现多页面分页 TabBarIOS/ViewPagerAndroid | 91 |
| 4.1.2 | 加载指示器——ActivityIndicator | 96 |

| | | |
|-------|-------------------------|-----|
| 4.1.3 | 地图——MapView | 97 |
| 4.1.4 | 渲染——Picker | 98 |
| 4.1.5 | 选择范围——Slider | 99 |
| 4.1.6 | 开关组件——Switch | 100 |
| 4.1.7 | 打开网页——WebView | 101 |
| 4.2 | 第三方组件 | 102 |
| 4.2.1 | react-native-swiper 的使用 | 103 |
| 4.2.2 | NativeBase 的使用 | 104 |
| 4.2.3 | NativeBase 如何解决跨平台问题 | 111 |
| 4.3 | 小结 | 113 |
| 第 5 章 | 原生平台的适配和调试 | 114 |
| 5.1 | iOS 平台的适配 | 114 |
| 5.1.1 | Images.xcassets 适配 | 115 |
| 5.1.2 | 自动布局 Auto Layout | 115 |
| 5.1.3 | Size Class 适配 | 116 |
| 5.2 | iOS 开发的调试技巧 | 117 |
| 5.3 | Android 平台的适配 | 118 |
| 5.3.1 | 适配原理 | 118 |
| 5.3.2 | 常用的适配属性 | 119 |
| 5.4 | Android 平台的调试技巧 | 122 |
| 5.5 | 小结 | 124 |
| 第 6 章 | React Native 的服务器端处理 | 125 |
| 6.1 | 学习 Node.js | 125 |
| 6.1.1 | 什么是 Node.js | 125 |
| 6.1.2 | 为什么选择 Node.js | 126 |
| 6.1.3 | 安装和使用 npm | 128 |
| 6.1.4 | Node.js 的开发流程 | 129 |
| 6.2 | 服务端接口的设计：RESTful | 132 |
| 6.3 | 实现电商 App 的服务器端接口 | 133 |
| 6.3.1 | Express 框架 | 133 |
| 6.3.2 | 查询商品接口 | 138 |
| 6.3.3 | 新建商品接口 | 142 |
| 6.3.4 | 更新商品接口 | 143 |
| 6.3.5 | 删除商品接口 | 144 |
| 6.4 | 网络前后端交互的原理 fetch | 145 |
| 6.5 | App 从服务器获取数据 | 146 |
| 6.5.1 | 获取商品信息 | 148 |
| 6.5.2 | 更新商品信息 | 151 |
| 6.5.3 | 新建商品 | 157 |
| 6.5.4 | 删除商品 | 158 |
| 6.6 | App 数据的本地化存储 | 160 |
| 6.6.1 | AsyncStorage 异步键值存储 | 160 |
| 6.6.2 | SQLite 数据库 | 164 |
| 6.6.3 | Realm 数据库 | 166 |

| | |
|---|------------|
| 6.7 小结 | 168 |
| 第 7 章 常用 React Native API | 169 |
| 7.1 屏幕设置相关 API | 169 |
| 7.1.1 获取屏幕宽高——Dimensions API | 170 |
| 7.1.2 获取屏幕分辨率——PixelRatio API | 173 |
| 7.2 动画 API | 174 |
| 7.2.1 RequestAnimationFrame API 帧动画 | 175 |
| 7.2.2 LayoutAnimation API 布局动画 | 177 |
| 7.2.3 Animated API 高级动画 | 179 |
| 7.3 组件、React Native API、原生平台 API | 184 |
| 7.3.1 组件和 API | 184 |
| 7.3.2 API 和原生平台 API | 184 |
| 7.4 实现自己的 Platform API | 185 |
| 7.4.1 支持 iOS 平台 | 186 |
| 7.4.2 支持 Android 平台 | 188 |
| 7.5 为应用添加更丰富的 API | 189 |
| 7.5.1 提示框和编辑框——AlertIOS | 190 |
| 7.5.2 前后台状态变化——AppState | 193 |
| 7.5.3 Android 物理“返回键”——BackAndroid | 195 |
| 7.5.4 日期和时间选择器——DatePickerAndroid/TimePickerAndroid | 196 |
| 7.5.5 基于位置的 Geolocation | 200 |
| 7.5.6 键盘事件——Keyboard | 202 |
| 7.5.7 设备联网状态——NetInfo | 204 |
| 7.5.8 权限设置——PermissionsAndroid | 205 |
| 7.5.9 悬浮提示框——ToastAndroid | 207 |
| 7.6 小结 | 208 |

第 3 篇 React Native 混合编程

| | |
|--|------------|
| 第 8 章 React Native 与原生平台混合编程（1） | 210 |
| 8.1 创建并移植项目 | 210 |
| 8.2 访问设备 | 211 |
| 8.2.1 访问 iOS 设备 | 213 |
| 8.2.2 访问 Android 设备 | 214 |
| 8.3 访问相册 | 217 |
| 8.3.1 读取 iOS 相册中的图片 | 219 |
| 8.3.2 读取 Android 相册中的图片 | 224 |
| 8.4 React Native 与原生平台的通信原理 | 228 |
| 8.5 React Native 平台调用原生页面 | 229 |
| 8.5.1 React Native 平台调用原生 iOS 页面 | 231 |
| 8.5.2 React Native 平台调用原生 Android 页面 | 234 |
| 8.6 原生平台调用 React Native 组件 | 238 |

| | | |
|---------------|---|------------|
| 8.6.1 | iOS 平台调用 React Native 组件 | 238 |
| 8.6.2 | Android 平台调用 React Native 组件 | 239 |
| 8.7 | 小结 | 240 |
| 第 9 章 | React Native 与原生平台混合编程 (2) | 241 |
| 9.1 | 使用相机拍摄图片 | 241 |
| 9.1.1 | 使用 iOS 相机拍摄 | 241 |
| 9.1.2 | 使用 Android 相机拍摄 | 244 |
| 9.2 | 添加图片选择提示框 | 247 |
| 9.2.1 | iOS 平台的提示 | 247 |
| 9.2.2 | Android 平台的提示 | 249 |
| 9.3 | 重构图片选择库 | 251 |
| 9.3.1 | iOS 平台的重构 | 251 |
| 9.3.2 | Android 平台的重构 | 253 |
| 9.4 | 向 iOS 项目中添加 React Native 支持 | 256 |
| 9.4.1 | 新建 iOS 项目 | 256 |
| 9.4.2 | 新建 React Native 项目 | 257 |
| 9.4.3 | 在 iOS 页面打开 React Native 组件 | 259 |
| 9.5 | 向 Android 项目中添加 React Native 支持 | 261 |
| 9.5.1 | 新建 Android 项目 | 261 |
| 9.5.2 | 新建 React Native 项目 | 261 |
| 9.5.3 | 在 Android 页面打开 React Native 组件 | 262 |
| 9.6 | 小结 | 264 |
| 第 10 章 | 电商 App 的复盘 | 265 |
| 10.1 | 电商 App 的文件 | 265 |
| 10.1.1 | JavaScript 文件 | 266 |
| 10.1.2 | iOS 原生代码文件 | 266 |
| 10.1.3 | Android 原生代码文件 | 267 |
| 10.2 | 电商 App 的结构 | 267 |
| 10.2.1 | Flexbox 的整体布局 | 268 |
| 10.2.2 | 应用的逻辑结构 | 268 |
| 10.2.3 | 应用的通信过程 | 269 |
| 10.3 | 优化和改进 | 270 |
| 10.3.1 | redux 是什么 | 270 |
| 10.3.2 | redux 代码示例 | 271 |
| 10.3.3 | redux 生态 | 274 |
| 10.4 | 用到的组件 | 275 |
| 10.5 | 小结 | 276 |

第 4 篇 App 的发布和更新

| | | |
|---------------|------------------------|------------|
| 第 11 章 | App 的发布 | 278 |
| 11.1 | App Store 苹果应用商店 | 278 |

| | | |
|--------|-------------------------|-----|
| 11.1.1 | 加入开发者计划 | 278 |
| 11.1.2 | 生成发布证书 | 280 |
| 11.1.3 | 注册 App ID | 283 |
| 11.1.4 | 生成描述文件 | 283 |
| 11.1.5 | 打包应用 | 284 |
| 11.1.6 | 发布到 App Store | 284 |
| 11.2 | Android 应用商店 | 285 |
| 11.2.1 | 生成签名文件 | 285 |
| 11.2.2 | 打包应用 | 287 |
| 11.2.3 | 发布到应用商店 | 288 |
| 11.3 | 小结 | 289 |
| 第 12 章 | App 的热部署 | 290 |
| 12.1 | 什么是热部署 | 290 |
| 12.2 | 解析 React Native 应用的工作原理 | 290 |
| 12.3 | 实现 React Native 的热部署 | 292 |
| 12.3.1 | 服务端实现 | 292 |
| 12.3.2 | 客户端实现 | 292 |
| 12.4 | 微软的热部署方案 CodePush | 295 |
| 12.4.1 | CodePush 简介 | 295 |
| 12.4.2 | CodePush 安装和注册 | 295 |
| 12.4.3 | 集成 CodePush SDK | 297 |
| 12.4.4 | 更改 iOS 应用 | 297 |
| 12.4.5 | 更改 Android 应用 | 301 |
| 12.5 | 小结 | 303 |
| 附录 A | ES 6 语法 | 304 |

React Native 入门和基础

第2章 全局解析 React Native 开发的基础技术

第 1 章 为什么要学习 React Native

无论读者是移动平台开发者，还是 Web 前端开发者，想必对现在“大红大紫”的 React Native 都有所耳闻。那么，除了“乘着 Facebook 这棵大树好乘凉”的优势之外，React Native 到底是何方“神圣”，有什么令大家“趋之若鹜”的优点呢？下面带着这样的好奇，来随本书一探究竟吧！

本章主要内容有：

- React Native 与 React.js 的对比。
- 为什么说 React Native 是跨平台的。
- React Native 应用的结构。
- React Native 的特点。
- React Native 的环境搭建。
- 创建第一个 React Native 应用。

1.1 看透 React Native

React Native (<http://facebook.github.io/react-native/>) 第一次进入公众的视野是在 2015 年 1 月的 React.js Conf(<http://conf.reactjs.org/>)上，随后，同年 5 月份，Facebook 在 F8 Conference(<https://www.fb8.com/>)上正式宣布：React Native 项目（如图 1.1 所示）在 Github 开源。结果一天之内，就收获了 5000 多颗星，受关注程度可见一斑！

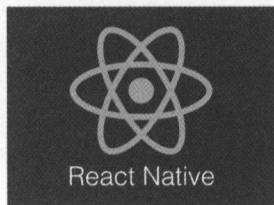



图 1.1 React Native Logo

 **小知识：**React.js Conf 是指 Facebook 的 React 开发者大会，F8 Conference 是指 Facebook 的开发者大会，Github 是全球最大的软件项目托管平台，也被戏称为“人类的代码仓库”。


1.1.1 React Native 与 React.js

想必读者在还没弄清 React Native 之前，又发现了一个“新朋友”React.js（下文简称 React），那到底什么是 React 呢？它和 React Native 又是什么关系呢？

先来看看 Facebook 官方（<https://code.facebook.com/projects/176988925806765/react/>）对 React 的定义：

React is a JavaScript library for building user interfaces

从上述官方的定义可以知道：React 是一个用于前端 UI 开发的 JavaScript 库。和其他类似的前端框架相比，例如，老牌的 Backbone (<http://backbonejs.org/>)、Google 推出的 Angular (<https://angularjs.org/>) 和以轻量级著称的 Vue.js (<http://cn.vuejs.org/>)，React 最大的不同是提出了 Virtual DOM（即虚拟 DOM）的设计，可以大大提升页面渲染的效率。


 **小知识：**移动平台开发很好理解，即移动平台上（例如 Apple 的 iOS 平台，Google 的 Android 平台）的软件开发，开发语言和技术主要有 Objective C、Swift 及 Java 等。而前端开发是相对于后端（又称服务器端）开发而言的，前端主要负责开发通过浏览器和用户交互的部分，开发语言和技术主要有 HTML、CSS 及 JavaScript 等。

但是，Facebook 不仅仅满足于 React 对前端开发技术的革新，又将 React 的设计移植到原生开发中，从而诞生了 React + Native 结合的产物，即 React Native。

虽然，React Native 刚开始只支持 iOS App 开发，但是从 2015 年 9 月起，React Native 也支持 Android App 开发，而且随着微软、三星等“IT 大佬”的加入，React Native 还将支持更多的移动平台，例如，Samsung 的 Tizen 平台 (<https://www.tizen.org/>)、Microsoft 的 Window Phone (<http://microsoft.github.io/code-push/articles/ReactNativeWindows.html>)。

所以，简单来说：

- React Native 和 React 使用了相同的开发语言 JavaScript 和相同的设计理念 React。
- React Native 和 React 运行的环境和平台却是完全不同的，React Native 是基于移动平台（如 iOS、Android 等），而 React 是基于浏览器。

 **提示：**国内网络环境下访问 React Native 官网 (<http://facebook.github.io/react-native/>) 可能较慢，读者可以访问国内的中文资源网站，例如 React Native 中文网 (<http://reactnative.cn/>)，或者自行搜索加快 React Native 官网访问速度的办法。


1.1.2 React Native 的跨平台

简单了解了 React Native 的由来之后，读者或许会有这样的疑问，开发移动平台 App 使用原生开发平台和语言就好了，为什么要出现使用 React Native 来开发移动平台 App 的技术呢？换句话说，React Native 到底可以解决什么问题呢？

在进一步讨论之前，笔者觉得有必要明确一下什么是原生应用和跨平台应用。

1. 原生应用

所谓原生应用是指：使用原生开发语言、工具和平台开发的应用。原生应用开发的优势在于拥有较高的平台成熟度，包括平台的稳定性、运行时的性能及完善的生态。

 **小知识：**所谓的“生态”应该算是比较抽象的概念，开发平台的生态圈包含了很多方面，从硬件上芯片和各种电子元器件的生产、供应，到软件上所使用的语言、开发工具及第三方开源库的数量质量，以及人的方面，如开发者的数量、水平等因素。

但是，原生应用开发也不是没有任何缺点，那就是开发成本较高，导致开发效率相对较低。例如，当一个产品需要支持多种类型的移动终端时，就需要熟悉多个原生平台开发的工程师。

2. 跨平台应用

为了解决产品满足多个平台的需求，就有了所谓的跨平台应用开发。根据实现跨平台方案的不同，也就有了以下几种常见的跨平台解决方案。

- 混合应用开发：在移动浏览器中嵌入 HTML 页面来开发移动应用，代表的有 Apache Cordova (<http://cordova.apache.org/>)，以及基于 Apache Cordova 衍生的 Inoic (<http://ionicframework.com/>) 等，如图 1.2 所示。
- 跨平台的语言：例如，基于 .NET 和 C# 的 Xamarin (<https://www.xamarin.com/>)，以及基于 Ruby 的 RubyMotion (<http://www.rubymotion.com/cn/>)，如图 1.3 所示。



图 1.2 Apache Cordova LOGO




图 1.3 Xamarin LOGO

- React Native：使用的是 Web 开发语言（JavaScript）和环境（Node.js）。除了本书介绍的 React Native 之外，类似的技术方案还有 NativeScript (<http://www.telerik.com/nativescript>)、Weex (<http://weex-project.io/>) 等，如图 1.4 所示。



图 1.4 Weex LOGO

 提示：想要了解关于更多 React Native 的架构和原理，可以参考 1.1.3 节。

1.1.3 解剖 React Native 应用的结构

在了解完这么多关于 React Native 的故事和优势之后，让我们走近 React Native，来进一步了解 React Native 的原理和架构。

React Native 应用的整体结构如图 1.5 所示。

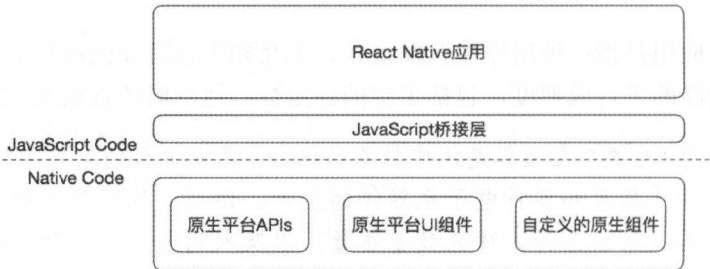


图 1.5 React Native 应用的整体结构

通过之前的介绍和图 1.5 可以看出：React Native 应用开发使用的是与 React 相同的开发语言 JavaScript 和设计思想 React，而底层仍然是基于原生平台的。这样，不同平台的适配就交由 React Native 平台去处理，而开发者只需要专注于 React Native 平台应用开发本身，体现出的优势如下。

- 应用层的开发变得简单、高效和跨平台。
- 应用稳定性、运行时的性能和原生平台接近。
- 在理解 React Native 原理之后，开发者也可以根据实际的产品需求开发自己的 React Native 组件，以复用已有原生平台的大量优秀组件。

1.2 React Native 的特点

那么，作为跨平台应用开发的“新贵”，React Native 相比其他跨平台技术到底有哪些优势呢？

1.2.1 其一：Learn Once, Write Anywhere

这句话是 React Native 官网 (<http://facebook.github.io/react-native/>) 对 React Native 的概述，简单明了地概括了 React Native 的最大特点和优点。

只需要学习 React Native 这一种开发方式（包括平台、语言和开发环境等）就可以开发多个不同平台的 App。

这句话简单来说就是 Learn Once, Write Anywhere，这也是 React Native 的宣传广告，如图 1.6 所示。



图 1.6 Learn Once, Write Anywhere 宣传广告

小知识：除了 React Native 提出的 Learn Once, Write Anywhere 的口号，Java 语言也提出过类似的口号 Write Once, Run Anywhere，两者看起来类似，但其实是完全不同的。React Native 就像上面介绍的，降低的是学习成本，针对不同平台可能还需要单独开发；而 Java 语言的意思是只需要开发一次，就可以成功运行在不同的平台和设备上。

目前，React Native 对 iOS、Android 平台的支持已经非常好了，在不远的将来，应该还会支持 Windows、Tizen 等更多的移动平台。

而且，React Native 的大多数组件也是可以在多个平台复用的，所以学习了 React Native 开发之后，完全可以胜任多个平台的开发任务且不需要很高的额外学习成本，大大降低了

开发成本。

1.2.2 其二：简单易学的开发语言

React Native 开发是基于 JavaScript 语言的，虽然 JavaScript 也是一门灵活、强大且复杂的语言，但是对于新人来说，上手速度相比 Objective-C 或 Java 等还是要快得多。而且，由于 JavaScript 严格模式的使用以及 ECMAScript 2015（下文简称 ES 6）标准的推出，JavaScript 被人诟病的各种语言问题也大大减少。

不仅如此，Facebook 为了进一步提高代码可读性和开发效率，还扩展出了 JSX 语法，即一种可以在 JavaScript 代码中直接书写 HTML 标签的语法糖。

例如，一个典型的 React Native 项目的 JavaScript 代码看起来是这样的：

```
01 export default class ch02 extends Component { // 每个页面可以理解成一个组件
02   render() {                                     // 渲染页面的函数
03     return (
04       <View style={styles.container}> // 页面根 View
05         <Text style={styles.welcome}>
06           Welcome to React Native!
07         </Text>
08         <Text style={styles.instructions}>
09           To get started, edit index.ios.js
10         </Text>
11         <Text style={styles.instructions}>
12           Press Cmd+R to reload,{'\n'}
13           Cmd+D or shake for dev menu
14         </Text>
15       </View>
16     );
17   }
16 }
```

除了开发语言使用 JavaScript 之外，在 React Native 开发中，样式和布局的技术相比原生平台也是比较简单的。

React Native 的样式使用了类似 CSS 的规范，只是根据 JavaScript 的语法要求将命名方式改成了“驼峰命名法”，例如，Web 开发中的 background-color 要写成 backgroundColor。

React Native 的布局使用了 Flexbox 布局方式，Flexbox 是 Flexible Box 的缩写，又称“弹性盒子布局”。Flexbox 布局不仅使用简单，最大的优势还在于提供了自适应显示设备和屏幕大小的能力，从而可以很好解决 iOS、Android 等屏幕适配问题。

例如，一个典型的 React Native 项目中的样式和布局代码看起来是这样的：

```
01 const styles = StyleSheet.create({
02   container: {                                     // 页面根 View 的样式
03     flex: 1,
04     justifyContent: 'center',
05     alignItems: 'center',
06     backgroundColor: '#F5FCFF'
07   },
08   welcome: {                                       // “欢迎”文本的样式
09     fontSize: 20,
10     textAlign: 'center',
11     margin: 10
12   },
```

```
13      instructions: {                                // “说明”文本的样式
14          textAlign: 'center',
15          color: '#333333',
16          marginBottom: 5
17      }
18  });
```

 提示：关于 JSX 和 Flexbox 布局的更多介绍，可以参考本书第 2 章的内容。

1.2.3 其三：接近原生应用的性能和体验

对于 React Native 上述的两个优点，混合应用开发的方式其实也都有，但是，混合应用开发的方式在实际开发中却存在性能和体验不佳的先天不足（其原理是在移动浏览器里嵌入 HTML 页面，导致原生平台的性能优势无法充分发挥出来）。

混合应用开发方式从 PhoneGap 发展到 Apache Cordova，而且衍生的 Ionic 也都在不断加强和优化性能，但是现阶段，在移动浏览器中嵌入 HTML 页面的运行效率，仍然无法和原生应用相媲美。

而 React Native 虽然使用的是类似混合应用开发的语言，但是其实现机制却完全不同：React Native 的底层仍然是基于原生平台的！所以，React Native 在性能和体验上与原生应用几乎没有太大差别，用户很难区分所使用的 App 到底是原生开发的还是 React Native 开发的。

1.2.4 其四：完善的生态系统

生态系统是衡量一个开发平台成熟度的重要标志，所以开发者在选择任何开发平台时，很有必要了解该平台的生态状况。

React Native 有着非常庞大的开发者社区和很高的活跃度，这点从 React Native 在 Github 上线第一天 5000 多颗星，截至 2017 年 1 月 4 日 40000 多颗星、9000 多次 Fork 以及 9000 多次提交就可以看出，如图 1.7 所示。

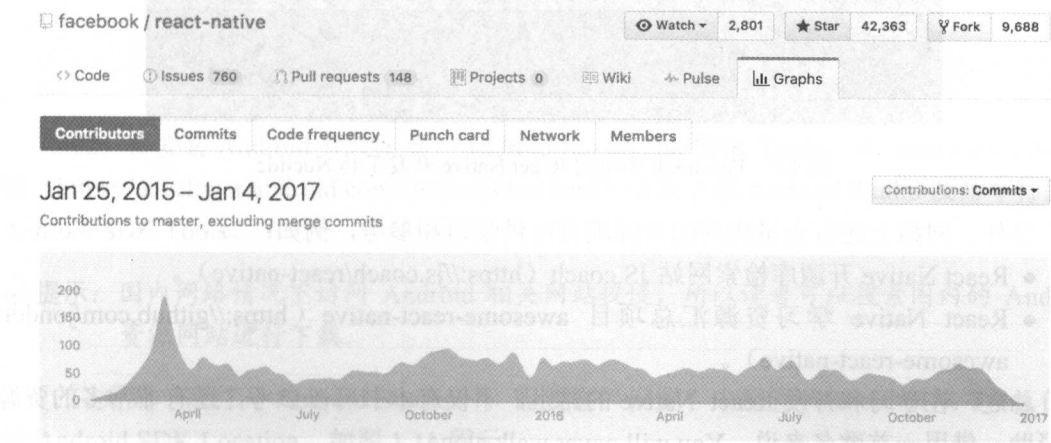


图 1.7 React Native 项目在 Github 上的关注度和贡献

Google Trends (<https://www.google.com/trends/explore?date=today%2012-m&q=ios%20development,android%20development,react%20native>) 也反映出了 React Native 开发的趋势和热度, 如图 1.8 所示。

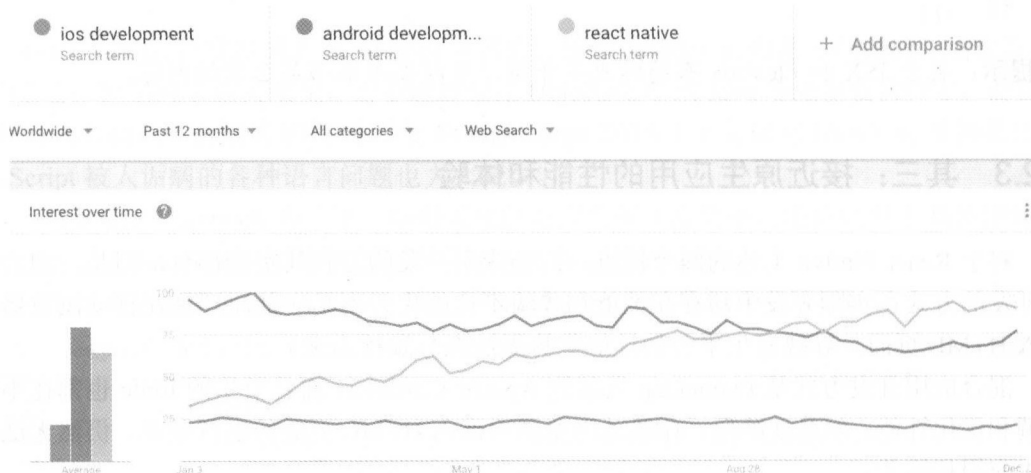


图 1.8 Google Trends 上 React Native 的趋势和热度

同时, Facebook 也在大力支持和推广 React Native, 并推出了官方的调试工具 React Developer Tools (<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>) 和开发工具 Nuclide (<https://nuclide.io/>), 如图 1.9 所示。

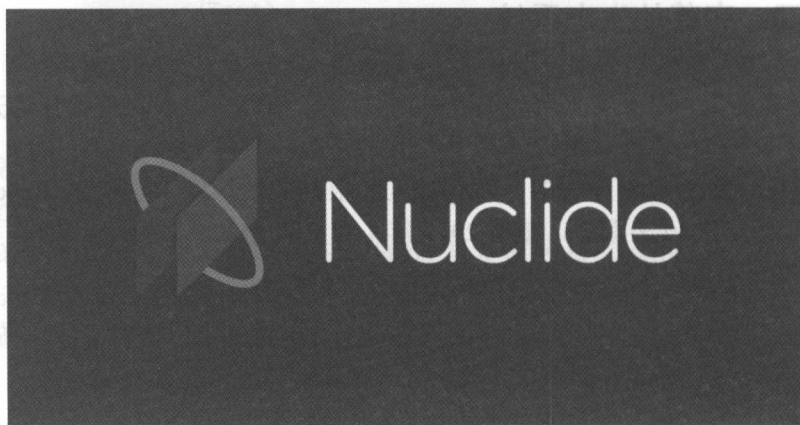


图 1.9 Facebook 推出的 React Native 开发工具 Nuclide

另外, 网络上还有大量优秀的开源项目可供学习和参考, 例如:

- React Native 开源库检索网站 JS.coach (<https://js.coach/react-native>)。
- React Native 学习资源汇总项目 awesome-react-native (<https://github.com/jondot/awesome-react-native>)。

总之, 在学习和开发 React Native 的路上, 不仅有本书可以参考, 还有非常多的资源和帮助, 借用一首歌名来说: You will never walk alone。

1.3 搭建 React Native 开发环境

“磨刀不误砍柴工”，在正式开发 React Native 应用之前，需要先搭建好 React Native 的开发环境。搭建 React Native 开发环境有以下几个主要步骤。

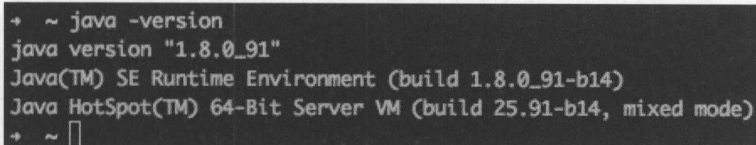
- 原生开发工具：iOS 开发使用 Xcode，Android 开发使用 Android Studio and SDK Tools。
- Node.js (<https://nodejs.org/>)：React Native 是借助 Node.js，即 JavaScript 运行时来创建 JavaScript 代码的。
- React Native (<https://www.npmjs.com/package/react-native/>)：安装 React Native 命令行工具。
- 其他辅助工具：代码编辑器 Nuclide (<https://nuclide.io/>)、远程调试工具 Chrome 浏览器 (<http://www.google.cn/chrome/browser/>) 等。

📌 注意：iOS 开发是依赖于 macOS 的，所以使用 React Native 开发 iOS 应用需要使用 macOS。

1.3.1 安装原生开发工具——Android

由于 React Native 应用仍然是基于原生平台（参考本书 1.1 节 React Native 的结构），所以搭建 React Native 的前提是安装原生开发工具。

(1) 安装 Java Development Kit (JDK)，从 JDK 官网 (<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>) 下载操作系统相应版本安装即可。安装成功后可以通过如图 1.10 所示的方法进行验证。



```
+ ~ java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)
+ ~
```

图 1.10 查看 JDK 安装是否成功及 Java 版本号

(2) 再安装 Android 开发工具：Android Studio and SDK Tools。从 Android Studio 官网 (<https://developer.android.com/studio/index.html>) 分别下载 Android Studio 及命令行工具 Android SDK Tools。

📌 提示：国内网络情况下访问 Android 相关网站较慢，所以读者可以搜索国内的 Android 资源网站进行下载。

(3) 第一次打开 Android Studio 时，需要在“设置”中配置 Android SDK Tools 的路径 Android SDK Location，如图 1.11 所示。

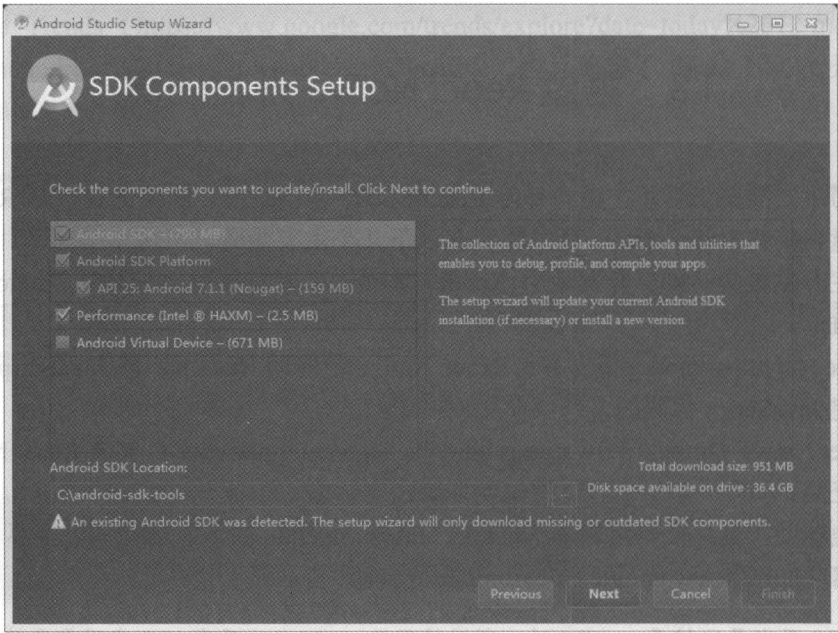


图 1.11 Android Studio 中配置 Android SDK Tools 路径

(4) 成功配置 Android SDK Tools 的路径之后，还要下载和安装 SDK 相关工具，如图 1.12 所示。

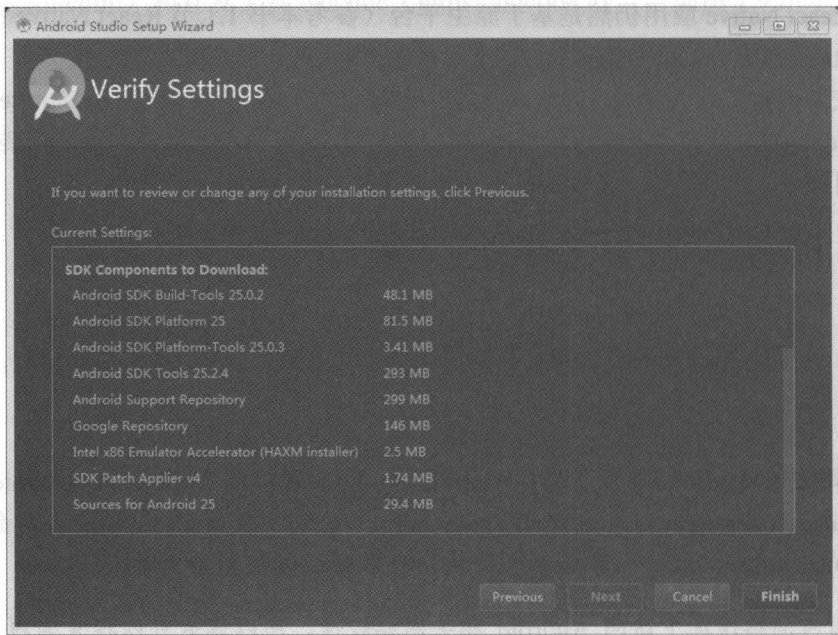


图 1.12 安装 Android SDK Tools 相关工具

(5) 此时，还需要将 Android SDK Tools 的路径加入到系统变量 PATH 中。

提示：无论是 JDK，还是 Android SDK Tools，以及以后要安装的其他开发工具（如 Node.js、npm、React Native 等），根据实际情况，都需要添加到系统变量 PATH

中，否则会发生找不到该工具或命令的错误。当然，有时候安装程序会自动完成该配置，请读者知悉。

对于 Linux 或 macOS 系统，将下面的配置添加到~/.bashrc 文件中。

```
export ANDROID_HOME=/path/to/android/sdk/tools
export PATH=${PATH}:${ANDROID_HOME}/tools
export PATH=${PATH}:${ANDROID_HOME}/platform-tools
```

对于 Windows 系统，将下面的路径添加到环境变量 PATH 中。

- Android SDK Tools 文件夹路径。
- Android SDK Tools 文件夹里的 tools 文件夹路径。
- Android SDK Tools 文件夹里的 platform-tools 文件夹路径。

添加效果如图 1.13 所示。

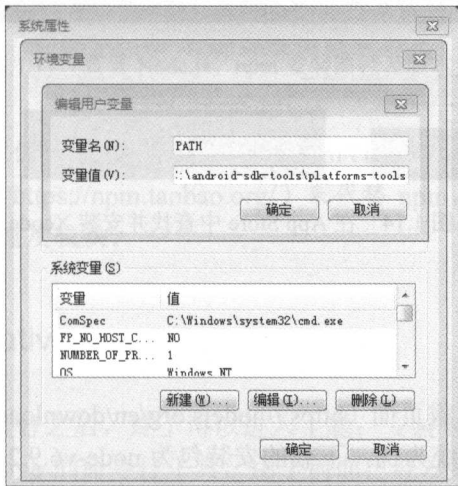


图 1.13 添加 Android SDK Tools 路径到 Windows 系统变量 PATH

提示：如果正确配置之后，还是找不到 Android SDK Tools 工具或命令，可以尝试重新加载环境变量或重启终端。

1.3.2 安装原生开发工具——iOS

首先需要再次提醒读者的是，使用 React Native 开发 iOS 应用是需要 macOS 的，所以经济条件允许的话，最好购置一台 Mac 电脑。只有在使用 React Native 同时开发 iOS 和 Android 应用，才能发挥出 React Native 跨平台的优势。

Xcode 的安装比较简单，在 macOS 中登录 Apple ID 后，打开 App Store 搜索 Xcode 安装即可，如图 1.14 所示。


注意：请务必在 Apple 官网或 App Store 下载 Xcode 安装程序。2015 年 9 月发生的轰动国内的 XcodeGhost 事件，起因就是因为开发者下载并使用了非官方被植入非法代码的 Xcode 安装程序。



图 1.14 在 App Store 中查找并安装 Xcode

1.3.3 安装 Node.js

打开 Node.js 官网的下载页面 (<https://nodejs.org/en/download/>)，下载当前系统对应的安装包，这里以 macOS 系统为例，下载的安装包为 node-v6.9.2.pkg。

提示：推荐读者使用最新的 LTS 版本，因为官方维护的周期较长，功能和稳定性较好。

- (1) 下载完成后双击安装包进行安装，如图 1.15 所示。
- (2) 根据安装向导提示，单击相应的“继续”或“同意”按钮，最后单击“安装”按钮进行安装。安装成功后，如图 1.16 所示。

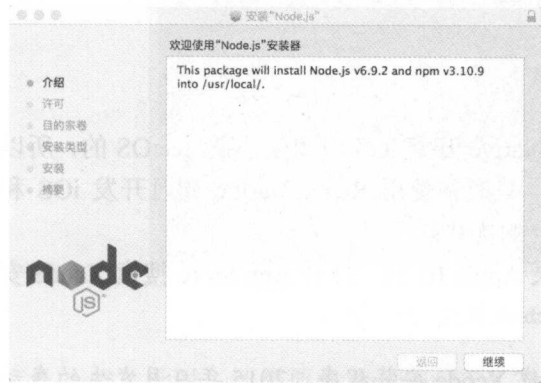


图 1.15 安装 Node.js

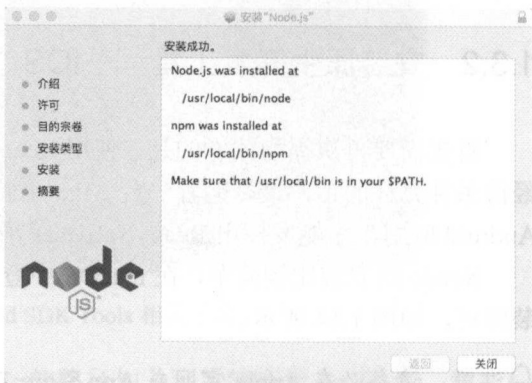


图 1.16 Node.js 安装成功

在图 1.16 中，需要注意以下有用的信息。

- Node.js 的安装路径: /usr/local/bin/node。
- npm 的安装路径: /usr/local/bin/npm。
- 保证/usr/local/bin 添加到了系统变量\$PATH 中。

提示: Node.js 在 Windows 系统上的安装过程和 macOS 相同, 并且 Node.js 的 Windows 安装包默认还会自动将安装好的工具和命令的路径添加到环境变量 PATH 中。

(3) 通过如图 1.17 所示方法验证安装是否成功。

```
+ ~ ls -l /usr/local/bin/node
-rwxrwxr-x 1 root wheel 30636560 12 7 03:00 /usr/local/bin/node
+ ~ ls -l /usr/local/bin/npm
lrwxr-xr-x 1 502 staff 38 12 28 14:25 /usr/local/bin/npm -> ../lib/node_modules/npm/bin/npm-cli.js
+ ~ node -v
v6.9.0
+ ~ npm -v
3.10.8
+ ~
```

图 1.17 查看 Node.js、npm 安装路径及版本号

小知识: 实际开发中通常使用 nvm(<https://github.com/creationix/nvm>)安装和管理 Node.js, 并且使用 cnpm (<https://npm.taobao.org/>) 来代替 npm, 这是因为 cnpm 使用的是淘宝源, 下载的速度较快。

1.3.4 安装 React Native

在 Node.js 及 npm 安装好之后, 终于可以进入主题: 安装 React Native 了。安装 React Native 就是安装 React Native 的命令行工具, 由于前面已经安装好 Node.js 及 npm, 安装 React Native 就非常简单了。使用如下命令即可:

```
npm install -g react-native-cli
```

提示: 如果使用 npm 下载速度较慢的话, 可以使用 cnpm 代替 npm, 即 cnpm install -g react-native-cli。

安装完成后, 可以通过如图 1.18 所示方法验证是否成功。

```
+ ~ react-native -h
Usage: react-native [command] [options]

Commands:
  init <ProjectName> [options]  generates a new project and installs its dependencies

Options:
  -h, --help    output usage information
  -v, --version  output the version number

+ ~ react-native -v
react-native-cli: 2.0.1
```

图 1.18 查看 react-native 命令行工具帮助和版本号

1.3.5 安装其他辅助工具

安装完以上所有工具后，理论上已经可以开始 React Native 开发之旅了，但在实际开发中，还有一些必备的高效生产工具，在这里推荐读者也安装一下。

1. Nuclide 开发工具

Nuclide (<https://nuclide.io/>) 是 Facebook 官方推出的一款 React Native 开发工具。从严格意义上说，Nuclide 并不是一款独立的编辑器，它只是基于 Atom (<https://atom.io/>) 的一个扩展，所以本书在后面的讨论中，并不严格区分 Nuclide 和 Atom。

当然，读者在了解 React Native 和 Nuclide 之前，肯定已经有自己熟悉的 JavaScript 的开发工具，如 Sublime Text (<http://www.sublimetext.com/>)、WebStorm (<https://www.jetbrains.com/webstorm/>) 等，但是笔者还是强烈推荐 Nuclide，原因如下。

- 官方出品，对 React Native 新特性及开发、调试支持更好。
- 基于 Atom，拥有了庞大的第三方插件库。

Nuclide 的安装过程如下：

- 从 Atom 官网 (<https://atom.io/>) 下载最新版本的 Atom。
- 打开 Atom，选择 Settings|Install Packages 命令，搜索并安装 Nuclide，如图 1.19 所示。

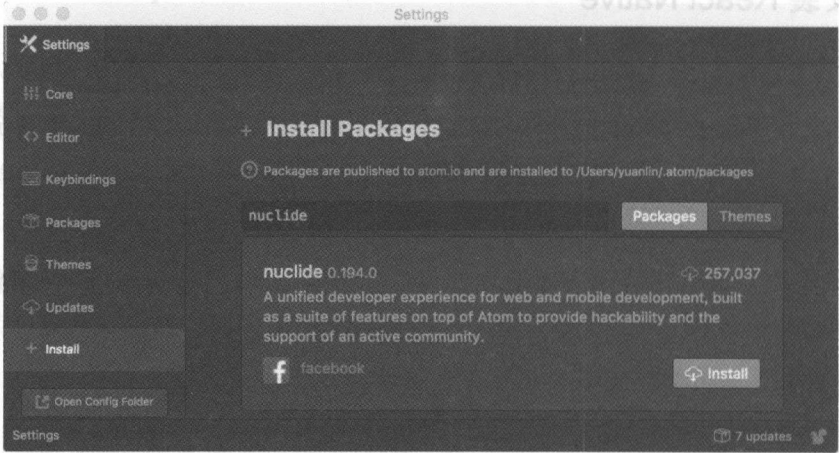


图 1.19 安装 Nuclide

2. Chrome 浏览器

Chrome 浏览器是 React Native 开发的远程调试工具，安装比较简单，这里就不介绍了。在 Chrome 浏览器安装完成后，还需要在 Chrome 的应用商店中安装这款 Chrome 插件：React Developer Tools，如图 1.20 所示。


 提示：和 Android 的问题一样，国内网络情况下访问 Chrome 相关网站也较慢，所以读者可以搜索国内的其他资源网站进行下载和安装。




图 1.20 安装 Chrome 插件 React Developer Tools

3. Watchman工具

Watchman (<https://facebook.github.io/watchman/docs/install.html>) 是由 Facebook 提供的监视文件系统变更的工具,它可以提高开发时的性能(捕捉文件的变化从而实现实时刷新)。

macOS 系统的安装比较简单,使用 HomeBrew (<http://brew.sh/>) 工具即可快速完成安装,安装命令如下:

```
brew update
brew install watchman
```

 小知识: HomeBrew 是 macOS 平台上的软件包管理工具,拥有安装、卸载、更新、查看和搜索软件包等很多实用的功能。

Linux 系统需要下载源码自己编译安装,方法也很简单,安装命令如下:

```
git clone https://github.com/facebook/watchman.git
cd watchman
git checkout v4.7.0 # the latest stable release
./autogen.sh
./configure
make
sudo make install
```

Windows 系统上的 Watchman 现在还处于 Alpha 内部测试阶段,对系统也有限制,例如,仅支持 Windows x64 on Windows Server 2012 R2 以及之后的最新 Windows 版本。如果读者想抢先体验的话,可以在 Watchman 项目的问题 19 (<https://github.com/facebook/watchman/issues/19>) 中找到详细的介绍。

1.4 第一个 React Native 应用

颇费一番周折搭建好环境之后，终于可以长舒一口气，来开发第一个 React Native 应用了。

1.4.1 初始化项目

首先，使用 React Native 命令行工具来初始化一个新的项目：

```
react-native init ch02
```

等待工程创建成功并安装好所有依赖后，使用 Atom 打开 ch02 项目，来仔细瞧一瞧 React Native 项目结构，如图 1.21 所示。

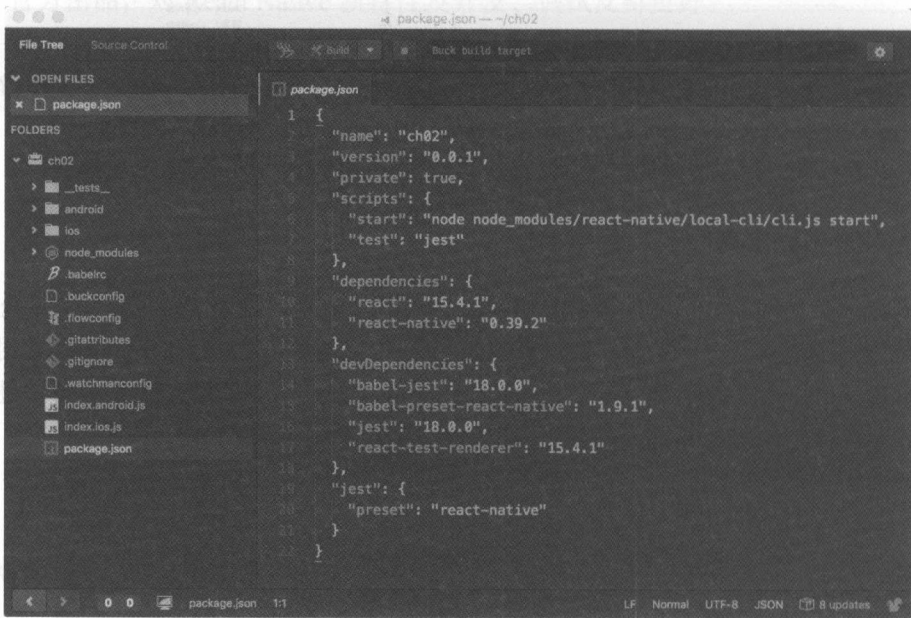


图 1.21 React Native 项目结构

其中目录和文件的详细说明如表 1.1 所示。

表 1.1 React Native 目录和文件的说明

| 目录/文件 | 说 明 |
|------------------|--|
| tests | React Native 工程单元测试文件夹 |
| android | 原生 Android 工程文件夹 |
| ios | 原生 iOS 工程文件夹 |
| node_modules | React Native 工程依赖的第三方库 |
| index.android.js | React Native 工程 Android App 入口文件 |
| index.ios.js | React Native 工程 iOS App 入口文件 |
| package.json | React Native 工程配置文件，描述了工程的所有信息以及第三方库的依赖关系，例如，刚才初始化的 ch02 工程依赖的 React Native 版本为 0.39.2 |

问题：为什么从网络下载的 React Native 项目无法直接运行？

回答：这是因为 .gitignore 文件中忽略了 node_modules 文件夹下面的文件，所以第三方库没有被添加到版本控制工具中，要运行从网络下载的 React Native 项目，首先需要在根目录下执行 npm install 或 cnpm install，安装所依赖的第三方库到 node_modules 文件夹中。

1.4.2 运行项目

成功运行 React Native 项目的前提是对应的原生开发工具安装和配置正确！例如：

- 运行 iOS App，需要正确安装和配置 Xcode 工具。
- 运行 Android App，需要正确安装和配置 Android Studio and SDK Tools。

1. 运行iOS App

首先，通过如下命令查看可用的 iOS 设备。

```
xcrun simctl list devices
```

接着，通过指定设备名称就可以运行 iOS App 了。

```
react-native run-ios --simulator "iPhone 7"
```

然后耐心地等待编译和安装成功后，第一个 React Native 应用就运行起来啦！如图 1.22 所示。

2. 运行Android App

同样，先通过如下命令查看可用的 Android 设备。

```
adb devices
```

接着，通过指定设备名称运行 Android App。

```
react-native run-android emulator-5554
```

运行效果如图 1.23 所示。

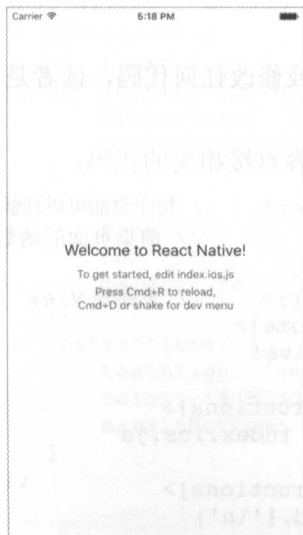


图 1.22 运行在 iOS 模拟器中的 React Native 应用

图 1.23 运行在 Android 模拟器中的 React Native 应用

运行完第一个 React Native 应用，想必读者已经感受到了 React Native 强大之处：不需要了解 iOS 和 Android 原生开发平台，使用 JavaScript 就可以开发出可以运行在多个平台的移动应用！

1.4.3 调试项目


在此，笔者还想补充说明一下关于 React Native 调试选项的相关内容。因为 React Native 不仅运行跨平台应用很方便，而且还提供了很多好用的调试工具加速开发。

真机调试时晃动设备就可以打开调试选项，模拟器调试时还可以使用如下快捷键。

- iOS 模拟器快捷键：command + D。
- Android 模拟器快捷键：command + M。

调试效果如图 1.24 所示。

这里打开 Enable Live Reload 选项，这样在 React Native 项目中做任何修改后，不需要重新启动或加载 App，运行中的 App 都可以自动更新了。

 **提示：**除了 Enable Live Reload 选项，React Native 还提供了其他很便利的调试选项，例如，远程调试选项 Debug JS Remotely，可以使用 Chrome 浏览器进行断点调试。关于更多调试选项的使用，读者可以在实际开发中探索和熟悉。

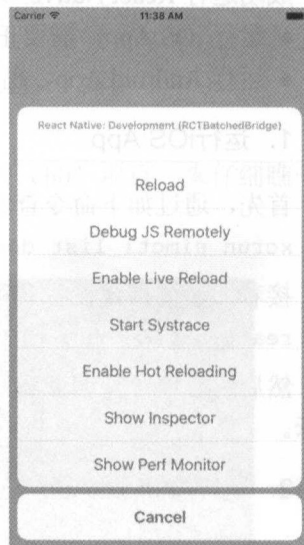


图 1.24 在模拟器中打开调试选项

1.5 小试牛刀——更改 React Native 项目源码

应用虽然已经运行起来了，但是到现在还没有看到或修改任何代码，读者是不是觉得意犹未尽呢？下面就来看看 React Native 项目的源码吧。

打开 index.ios.js 文件，可以看到与显示在设备上内容直接相关的代码：

```
01 export default class ch02 extends Component { // 每个页面可以理解成一个组件
02     render() {                                // 渲染页面的函数
03         return (
04             <View style={styles.container}>    // 页面根 View
05                 <Text style={styles.welcome}>
06                     Welcome to React Native!
07                 </Text>
08                 <Text style={styles.instructions}>
09                     To get started, edit index.ios.js
10                 </Text>
11                 <Text style={styles.instructions}>
12                     Press Cmd+R to reload,{'\n'}
13                     Cmd+D or shake for dev menu
14                 </Text>
05             </View>
06         )
07     }
08 }
```

```


15         </View>
16     );
17 }
16 }

```

为了证实我们的想法，将代码中的文本内容从“Welcome to React Native!”修改为“我的第一个 React Native 应用！”，然后在 iOS 模拟器中使用快捷键 `command + R` 重新加载应用，果然界面更新了！如图 1.25 所示。



图 1.25 修改文本内容后的效果

 **提示：**如果已经打开了 Enable Live Reload 调试选项，就不需要手动重新加载应用了，修改完代码直接可以看到效果。

接着，再看看显示样式的代码：

```

01 const styles = StyleSheet.create({
02     container: {                                // 页面根 View 的样式
03         flex: 1,
04         justifyContent: 'center',
05         alignItems: 'center',
06         backgroundColor: '#F5FCFF'
07     },
08     welcome: {                                    // “欢迎”文本的样式
09         fontSize: 20,
10         textAlign: 'center',
12         margin: 10
12     },
13     instructions: {                               // “说明”文本的样式
14         textAlign: 'center',
15         color: '#333333',
16         marginBottom: 5
17     }
18 });

```


在 `welcome` 样式中添加 `color: 'red'` 属性:

```
01 welcome: {
02   fontSize: 20,
03   textAlign: 'center',
04   margin: 10,
05   color: 'red' // 也可以用 RGB 值 '#FF0000' 来表示红色
06 },
```

🔔 注意: 第一次编写 React Native 代码时很容易发生遗漏逗号 “,” 等拼写错误。

重新加载应用后, 效果如图 1.26 所示。

以上是 iOS App 的运行效果, 同样也可以对 `index.android.js` 文件做类似的修改, 重新加载 Android App 效果如图 1.27 所示。



图 1.26 修改文字颜色后效果



图 1.27 Android App 运行的效果

🔔 提示: 实际开发中, `index.ios.js` 和 `index.android.js` 往往复用相同的逻辑, 即将相同的代码提取到公共文件中, 这样就可以大大发挥 React Native 的跨平台优势。

1.6 小 结

React Native 兼顾了开发的难易度、稳定性、性能、成本以及复用等产品开发中的诸多因素, 再加上 React 以及 React Native 自身优秀的设计及开源社区的积极参与和贡献, React Native 才有了今天这样的地位。

通过本章的介绍, 想必读者已经对 React Native 开发有了一个初步的认识, 体会到了 React Native 开发简单、跨平台的优势。接下来将通过开发一个完整的 App 实例, 进一步学习和掌握 React Native。

第2章 全局解析 React Native 开发的基础技术

在第1章搭建好 React Native 开发环境之后，我们开发了第一个 React Native 应用。虽然其功能比较简单，但却向着学习 React Native 开发迈开了一大步，意味着读者能够独立完成一个 React Native 应用的开发。

从本章开始，我们将从零开始开发一个功能更加完备、强大的 React Native 应用。还在等什么？赶紧进入状态吧！


本章主要内容有：

- 掌握版本控制工具 Git 的使用。
- 了解 JSX 解决方案。
- 熟悉 React Native 的布局。
- 了解如何调试 React Native 项目。
- 从零开始设计第一个完成的电商 App。

2.1 开发具备的基础知识说明


日常生活中，人们越来越离不开的就是网购，因此，本书就以典型的电商类移动应用为例，向读者展示使用 React Native 设计、开发应用的全过程。

不过，在正式开发电商类移动应用之前，有必要先了解一些 React Native 开发的基础知识。

 **提示：**本书面向对 JavaScript 有一定了解的读者，所以 JavaScript 相关知识本书不做深入解释，想要了解更多 JavaScript 知识，请读者参考相关书籍和教程。

React Native 开发中需要具备的基础知识如下。

- **Git：**最流行的版本控制工具，是开发中代码管理的基础。
- **JSX：**React Native 开发所使用的语言，一种基于 JavaScript 的扩展语法。
- **Flexbox 布局：**React Native 开发的布局技术，是 UI (User Interface) 开发的核心。
- **调试：**提高 React Native 开发效率的重要手段。

 **小知识：**UI (User Interface) 指用户界面（亦称使用者界面），是系统和用户之间进行交互和信息交换的媒介，它实现信息的内部形式与人类可以接受形式之间的转换。

2.2 Git 版本控制工具

对于现在的软件项目来说，版本控制工具应该是“标配”的开发工具之一了。

问题：什么是版本控制工具？

回答：版本控制工具提供完备的版本管理功能，用于存储、追踪目录（文件夹）和文件的修改历史。

2.2.1 安装 Git

这里笔者推荐一款免费、开源、简单易用的版本控制工具 Git (<https://git-scm.com/>)。

 小知识：Git 的诞生与 Linux 有不解之缘，Git 是由被誉为“Linux 之父”的 Linus Torvalds (<https://zh.wikipedia.org/wiki/%E6%9E%97%E7%BA%B3%E6%96%AF%C2%B7%E6%89%98%E7%93%A6%E5%85%B9>) 最初开发的，他认为之前现有的版本控制工具，例如 CVS (<http://www.nongnu.org/cvs/>)、SVN (<https://subversion.apache.org/>) 都满足不了 Linux Kernel 开发的需求（免费、简单、高效以及分布式），所以就决定自己开发一款全新的版本控制工具 Git。

Git 的安装比较简单，请读者自行到官网下载页面 (<https://git-scm.com/downloads>) 下载操作系统的相应版本安装即可。安装成功后可以通过如图 2.1 所示方法进行验证。

```
+ ~ git version
git version 2.9.3 (Apple Git-75)
+ ~ git --help
usage: git [--version] [--help] [-C <path>] [-c name=value]
       [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
       [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
       [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
       <command> [<args>]
```

图 2.1 查看 Git 版本号和帮助来验证 Git 安装是否成功

2.2.2 Git 常用命令

本节来了解 Git 常用命令的用法。

(1) 新建一个文件夹，然后在新建的文件夹中创建 Git 仓库。使用的命令如下：

```
mkdir git-demo          // 新建 git-demo 文件夹
cd git-demo             // 进入 git-demo 文件夹
git init                // 创建了一个新的 Git 仓库
```


效果如图 2.2 所示。

```
+ ~ mkdir git-demo
+ ~ cd git-demo
+ git-demo git init
Initialized empty Git repository in /Users/yuanlin/git-demo/.git/
+ git-demo git:(master) []
```

图 2.2 新建本地 Git 仓库

(2) 在刚才新建的 Git 仓库中，就可以进行添加和提交修改的操作了。

```
touch test.file // 为了演示 Git 的使用，这里先新建 test.file 文件
```


 **提示：** touch 命令是 Linux 和 macOS 系统下的命令行工具，touch test.file 的作用是在当前目录下新建空文件 test.file，读者也可以使用其他自己熟悉的方法新建测试文件。

(3) Git 添加修改的命令使用方法如下：

```
git add test.file // 添加 test.file 文件
git add * // 添加所有文件
```

(4) Git 提交修改的命令使用方法如下：

```
git commit -m "新建 test.file" // 提交修改并且描述此次修改的内容
```

 **提示：** 第一次使用 git commit 命令时，会提示用户配置 Git 账户和邮箱，配置方法为 git config --global user.name "Your Name"、git config --global user.email "Your Email"。

此时，Git 的工作流如图 2.3 所示。

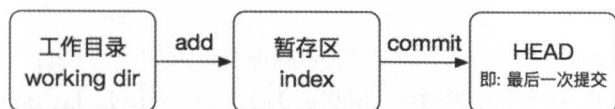


图 2.3 Git 添加和提交修改的工作流

(5) 添加和提交操作成功之后，可以通过如下方法查看结果。

```
git log // 查看 Git 提交的历史记录
```

此时，对于上面的提交，读者可以看到类似下面的信息。

```
commit COMMIT-ID
Author: GIT-USER-NAME <GIT-USER-EMAIL>
Date: COMMIT-DATE
    新建 test.file
```

(6) 除了自己创建 Git 仓库，还可以从网上下载已有的 Git 仓库代码。这里，以 Github 上 React Native 项目 (<https://github.com/facebook/react-native>) 为例，使用的命令如下：

```
git clone https://github.com/facebook/react-native // 将远程仓库复制到本地
```

(7) 此时，由于 Git 仓库是在远程服务器上，还需要用到 git pull 和 git push 这两个命令来操作 Git 仓库。


```
cd react-native // 首先需要进入 Git 仓库所在的文件夹
```

把刚才复制到本地的 Git 仓库更新到远程仓库的最新改动，使用 git pull 命令。

```
git pull
```

把刚才复制到本地的 Git 仓库提交的修改提交到远程仓库中，使用 git push 命令。


```
git push
```

提示：使用 Github 上 React Native 项目的例子，执行 git push 可能会提交失败，这是因为提交至远程仓库需要权限，请读者知悉。

当然，除了上述介绍的基本用法之外，Git 命令还有很多，举例如下。


- git status: 查看 Git 仓库状态。
- git diff: 查看 Git 仓库修改内容的差异。
- git branch: 使用和管理 Git 分支。
- git tag: 使用和管理 Git 标签。

本书限于篇幅就不一一介绍了，想要深入了解的读者可以参考 Git 相关书籍和教程。

提示：如果读者对 Git 命令不熟悉的话，推荐使用 Git 的图形化工具，例如，SourceTree (<https://www.sourcetreeapp.com/>) 或者 Tower (<https://www.git-tower.com/mac/>)，它们都提供了 Windows 和 macOS 的版本。

2.3 React Native 的 JSX 解决方案

JSX 并不是一门新的开发语言，而是 Facebook 提出的语法方案：一种可以在 JavaScript 代码中直接书写 HTML 标签的语法糖，所以，JSX 本质上还是 JavaScript 语言。

小知识：语法糖 (Syntactic sugar) 是由英国计算科学家彼得·兰丁 (<https://zh.wikipedia.org/wiki/%E5%BD%BC%E5%BE%97%C2%B7%E5%85%B0%E4%B8%81>) 发明的一个术语，指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。语法糖让程序更加简洁，有更高的可读性。

在 React 和 React Native 开发中，不一定非要使用 JSX，也可以直接使用 JavaScript 进行开发。但是，强烈建议读者使用 JSX！因为 JSX 在定义类似 HTML 这种树形结构时，简单明了，极大地提高了开发和维护的效率。

下面以 1.4 节第一个 React Native 应用中的代码为例：

```
01 export default class ch02 extends Component { // 每个页面可以理解成一个组件
02     render() {                                // 渲染页面的函数
03         return (
04             <View style={styles.container}>    // 页面根 View
05                 <Text style={styles.welcome}>
06                     Welcome to React Native!
07                 </Text>
08                 <Text style={styles.instructions}>
09                     To get started, edit index.ios.js
10                 </Text>
11                 <Text style={styles.instructions}>
12                     Press Cmd+R to reload,{'\n'}
13                     Cmd+D or shake for dev menu
14                 </Text>
15             </View>
16         );
17     }
18 }
```

在上述代码中，组件的 `render()` 方法函数是用于渲染页面的，它的返回值是一个 `View` 的对象，但是为什么没有发现创建对象和设置属性的代码呢？原来，`JSXTransformer` 帮我们把代码中 XML-Like 语法编译转换成真实可用的 JavaScript 代码，它不仅仅创建 `View` 对象、设置 `View` 样式和布局，同时更加贴心的是，还构建了 `View` 之间的树形结构。例如，上述例子中的树形结构是这样的：

```
Root View (style container)
---- Sub Text 1 (style welcome)
---- Sub Text 2 (style instructions)
---- Sub Text 3 (style instructions)
```

2.4 React Native 的 Flexbox 布局

无论是在移动平台还是 Web 前端开发中，布局技术都是必不可少的。了解 Web 开发的读者想必都听说过著名的 CSS “盒子模型”，如图 2.4 所示。

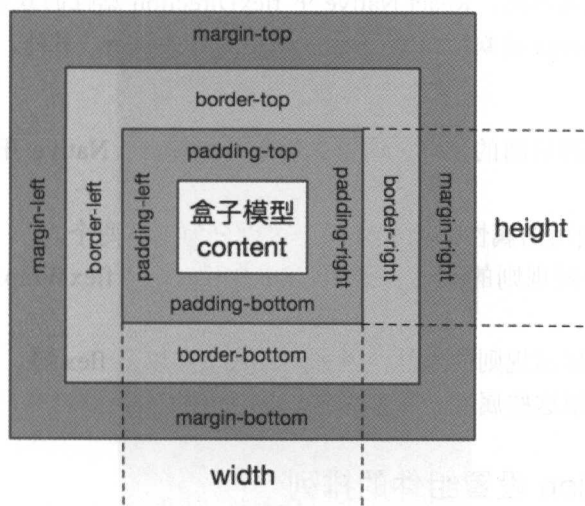


图 2.4 CSS “盒子模型”

CSS “盒子模型”依赖于 `position` 属性、浮动属性以及 `display` 属性来进行布局，所以，对于一些特殊但常用的布局（例如垂直居中）实现就比较困难。

于是，在 2009 年，W3C 提出了一种新的方案——Flexbox 布局。Flexbox（Flexible Box 的缩写，又称弹性盒子布局）布局旨在提供一个更加有效的方式制定、调整 and 分布一个容器里的项目布局，即使他们的大小是未知或者动态的。Flexbox 布局的主要思想是让容器有能力让其子项目能够改变其宽度、高度（甚至顺序），以最佳方式填充可用空间（主要是为了适应所有类型的显示设备和屏幕大小）。

小知识：W3C（World Wide Web Consortium）指万维网联盟，该组织建立于 1994 年，其宗旨是通过促进通用协议的发展并确保其通用型，以激发 Web 世界的全部潜能。

目前，主流浏览器都已经很好地支持 Flexbox 布局，如图 2.5 所示。

| IE | Edge * | Firefox | Chrome | Safari | Opera | iOS Safari * | Opera Mini * | Android Browser * | Chrome for Android |
|----|--------|---------|--------|--------|-------|--------------|--------------|-------------------|--------------------|
| | | | 49 | | | | | 4.3 | |
| | | | 51 | | | | | 4.4 | |
| | | | 54 | | | 9.3 | | 4.4.4 | |
| 11 | 14 | 50 | 55 | 10 | 42 | 10.1 | all | 53 | 55 |
| | 15 | 51 | 56 | TP | 43 | | | | |
| | | 52 | 57 | | 44 | | | | |
| | | 53 | 58 | | | | | | |

图 2.5 Flexbox 布局与主流浏览器的兼容性

React Native 实现了 Flexbox 布局的大部分功能，并且在实际应用开发中也使用 Flexbox 来实现布局。这不仅使 React Native 的 UI 开发变得更加简单，还很好地解决了 iOS、Android 等屏幕适配的问题。

提示：React Native 中 Flexbox 布局的工作原理和 Web 开发基本一致，只有少许差异。例如，默认值不同，React Native 中 flexDirection 属性的默认值是 column 而不是 row，alignItems 的默认值是 stretch 而不是 flex-start，另外，flex 只能指定一个数字值。

为了方便读者理解后面的代码，这里先简单介绍 React Native 开发中 Flexbox 布局的使用。

Flexbox 布局所使用的属性，简单来说，可以分为以下两个。

- 决定子组件排列规则的属性，例如，flexDirection、flexWrap、justifyContent 以及 alignItems 等。
- 决定组件自身显示规则的属性，例如，alignSelf 以及 flex 等。

下面分别简单介绍这些属性。

2.4.1 flexDirection 设置组件的排列

flexDirection 属性表明组件中子组件的排列方向，取值有 column（默认值）、row 以及 row-reverse。


例如，在不设置 flexDirection 的情况下，下面代码中的子组件 view1 和 view2 是按照默认值 column 纵向排列的，代码如下：

```
01 export default class flexbox extends Component {
02   render() {
03     return (
04       <View style={styles.container}> // 页面根 View
05         <View style={styles.view1}></View> // 子组件 view1
06         <View style={styles.view2}></View> // 子组件 view2
07       </View>
08     );
09   }
10 }
11
12 const styles = StyleSheet.create({
```

```

13     container: {
14         flex: 1, backgroundColor: 'gray',
15     },
16     view1: {                                //子组件 view1 的样式
17         height: 150,
18         width: 150,
19         backgroundColor: 'red'
20     },
21     view2: {                                //子组件 view2 的样式
22         height: 150,
23         width: 150,
24         backgroundColor: 'green'
25     }
26 });

```

 **提示：**为了节约篇幅，上述例子中只附上了关键代码，例如导入模块或组件的代码，形如 `import * from *`。但是读者在实际开发过程中，千万不要忘记先导入相关模块或组件，否则会发生错误。

效果如图 2.6 所示。

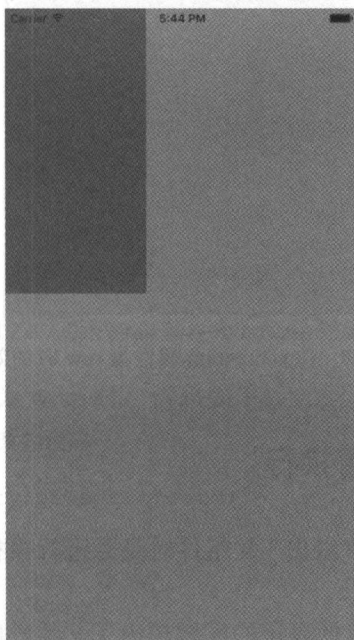


图 2.6 flexDirection 属性为 column 时的效果

当将 flexDirection 属性设置为 row 时，代码如下：

```

01 // 这里省略了没有修改的代码
02
03 const styles = StyleSheet.create({
04     container: {
05         flex: 1, backgroundColor: 'gray',
06         flexDirection: 'row'
07     },
08     view1: {                                //子组件 view1 的样式
09         height: 150,
10         width: 150,

```

```
11     backgroundColor: 'red'
12   },
13   view2: {                                //子组件 view2 的样式
14     height: 150,
15     width: 150,
16     backgroundColor: 'green'
17   }
18 });
```

那么，子组件 view1 和 view2 将按照 row 进行横向排列，效果如图 2.7 所示。

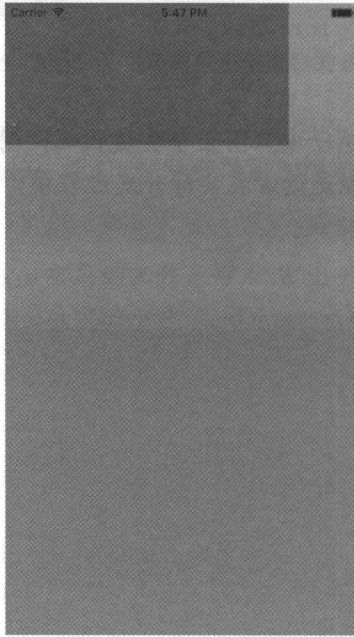


图 2.7 flexDirection 属性为 row 时的效果

2.4.2 flexWrap 设置是否换行

flexWrap 属性表明子组件“溢出”父组件时是否进行换行，取值有 nowrap（默认值）、wrap 以及 wrap-reverse。

这里，所谓的溢出是指子组件显示的区域超出了父组件的空间。为了模拟溢出的效果，下面修改 view1 和 view2 的大小，代码如下：

```
01 // 这里省略了没有修改的代码
02
03 const styles = StyleSheet.create({
04   container: {
05     flex: 1, backgroundColor: 'gray',
06     flexDirection: 'row'
07   },
08   view1: {
09     height: 200,
10     width: 200,                                // 增大了 view1 的宽度
11     backgroundColor: 'red'
12   },
13   view2: {
```

```

14     height: 200,
15     width: 200,           // 增大了 view2 的宽度
16     backgroundColor: 'green'
17   }
18 });

```

由于没有设置 `flexWrap`, 所以当 `view2` 显示不全时, 会按照默认值 `nowrap` 不进行换行, 效果如图 2.8 所示。

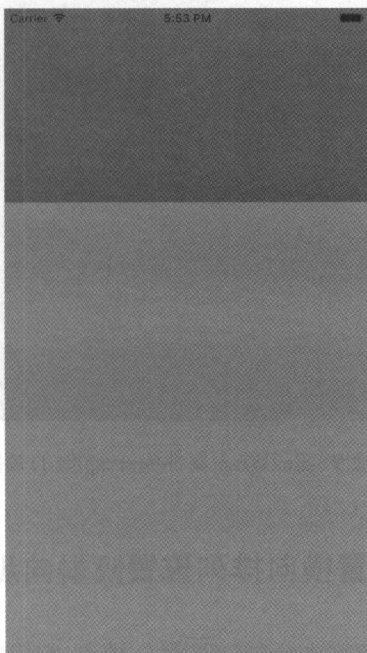


图 2.8 `flexWrap` 属性为 `nowrap` 时的效果

当将 `flexWrap` 属性设置为 `wrap` 时, 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 const styles = StyleSheet.create({
04   container: {
05     flex: 1, backgroundColor: 'gray',
06     flexDirection: 'row',
07     flexWrap: 'wrap'
08   },
09   view1: {
10     height: 200,
11     width: 200,
12     backgroundColor: 'red'
13   },
14   view2: {
15     height: 200,
16     width: 200,
17     backgroundColor: 'green'
18   }
19 });

```

此时, 由于 `view2` 和 `view1` 的总宽度 ($200 + 200 = 400$) 大于屏幕的宽度 (使用的 iPhone 7 宽度=375), 所以 `view2` 将会换行显示, 效果如图 2.9 所示。

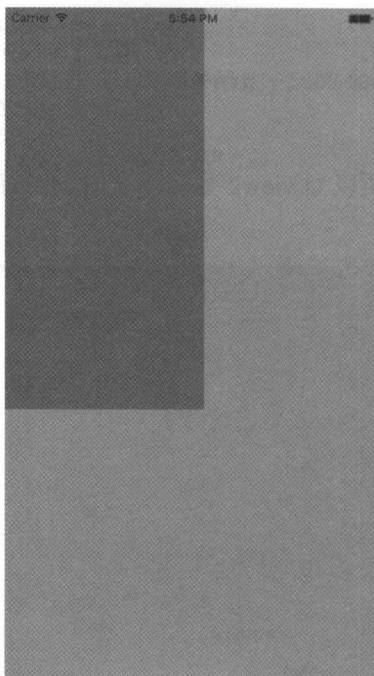


图 2.9 flexWrap 属性为 wrap 时的效果

2.4.3 justifyContent 设置横向排列位置

justifyContent 属性表明组件中子组件横向排列在其父容器的哪个位置，取值有 flex-start、flex-end、center、space-between 以及 space-around。

例如，想要实现子组件水平居中的效果，就可以将 justifyContent 的值设置为 center，代码如下：

```
01 // 这里省略了没有修改的代码
02
03 const styles = StyleSheet.create({
04   container: {
05     flex: 1, backgroundColor: 'gray',
06     flexDirection: 'row',
07     justifyContent: 'center'
08   },
09   view1: {
10     height: 150,
11     width: 150,
12     backgroundColor: 'red'
13   },
14   view2: {
15     height: 150,
16     width: 150,
17     backgroundColor: 'green'
18   }
19 });
```

效果如图 2.10 所示。

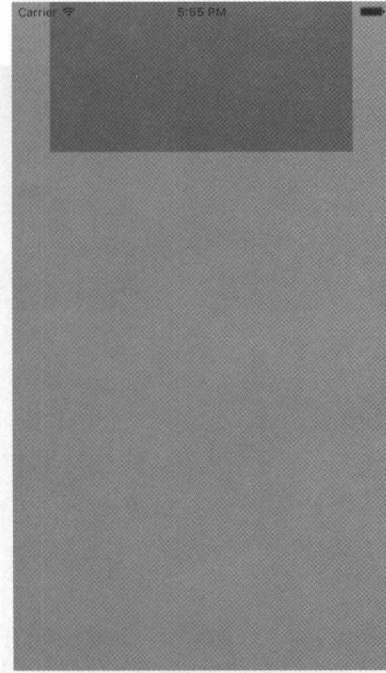


图 2.10 justifyContent 属性为 center 时的效果

2.4.4 alignItems 设置纵向排列位置

alignItems 属性表明组件中子组件纵向排列在其父容器的哪个位置，取值有 flex-start、flex-end、center、baseline 以及 stretch。

alignItems 属性和 justifyContent 的作用相似，只是 justifyContent 决定了子组件横向排列的位置，而 alignItems 决定了子组件纵向排列的位置。

如果想要实现子组件垂直居中的效果，那么就可以将 alignItems 的值设置为 center，代码如下：

```

01 // 这里省略了没有修改的代码
02
03 const styles = StyleSheet.create({
04   container: {
05     flex: 1, backgroundColor: 'gray',
06     flexDirection: 'row',
07     alignItems: 'center'
08   },
09   view1: {
10     height: 150,
11     width: 150,
12     backgroundColor: 'red'
13   },
14   view2: {
15     height: 150,
16     width: 150,
17     backgroundColor: 'green'
18   }
19 });

```


效果如图 2.11 所示。

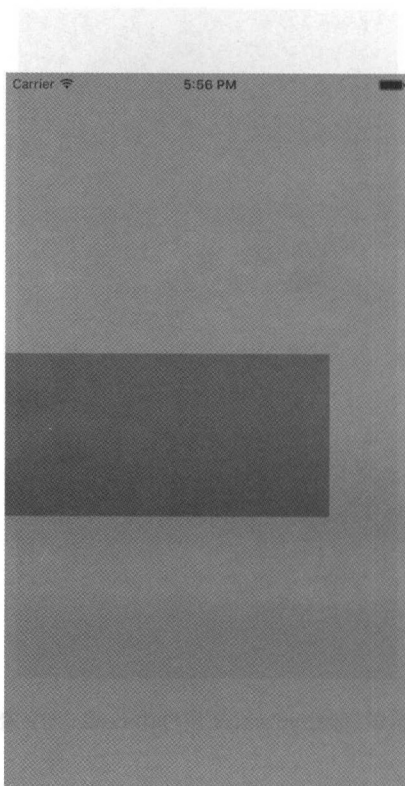


图 2.11 alignItems 属性为 center 时的效果

在了解了 justifyContent 和 alignItems 属性的用法之后，想必读者很容易就能够自己实现水平垂直居中的效果了，代码如下：

```
01 // 这里省略了没有修改的代码
02
03 const styles = StyleSheet.create({
04   container: {
05     flex: 1, backgroundColor: 'gray',
06     flexDirection: 'row',
07     justifyContent: 'center',
08     alignItems: 'center'
09   },
10   view1: {
11     height: 150,
12     width: 150,
13     backgroundColor: 'red'
14   },
15   view2: {
16     height: 150,
17     width: 150,
18     backgroundColor: 'green'
19   }
20 });
```

水平垂直居中的效果如图 2.12 所示。

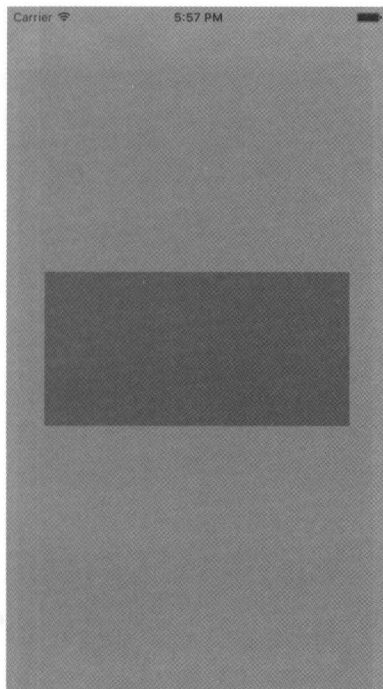


图 2.12 使用 `justifyContent` 和 `alignItems` 属性实现水平垂直居中效果

2.4.5 `alignSelf` 设置特定组件的排列

`alignSelf` 属性表明某个特定组件的排列情况，取值有 `auto`、`flex-start`、`flex-end`、`center` 以及 `stretch`。

这里以 `center` 为例，代码如下：

```
01 // 这里省略了没有修改的代码
02
03 const styles = StyleSheet.create({
04   container: {
05     flex: 1, backgroundColor: 'gray',
06   },
07   view1: {
08     height: 150,
09     width: 150,
10     backgroundColor: 'red'
11   },
12   view2: {
13     height: 150,
14     width: 150,
15     backgroundColor: 'green',
16     alignSelf: 'center'
17   }
18 });
```

效果如图 2.13 所示。

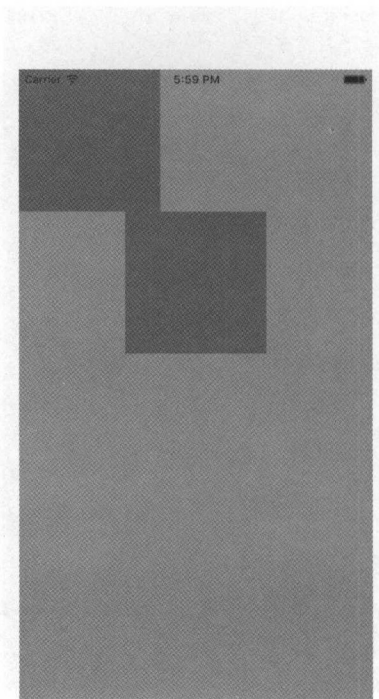


图 2.13 alignSelf 属性为 center 时的效果

2.4.6 flex 设置组件所占空间

flex 属性可以让组件动态计算和配置自己所占用的空间大小，取值是数值。

如果想要 view1 和 view2 填满父组件，并且 view1 和 view2 的大小相同，使用 flex 就很容易实现，代码如下：

```
01 // 这里省略了没有修改的代码
02
03 const styles = StyleSheet.create({
04   container: {
05     flex: 1
06   },
07   view1: {
08     flex: 1,
09     backgroundColor: 'red'
10   },
11   view2: {
12     flex: 1,
13     backgroundColor: 'green'
14   }
15 });
```

效果如图 2.14 所示。

flex 属性使用如此简单，但表现力却非常强大，它是 Flexbox 布局实现自适应设备和屏幕尺寸的核心，读者需要在后面的 React Native 开发中逐步熟练掌握 flex 属性的使用。

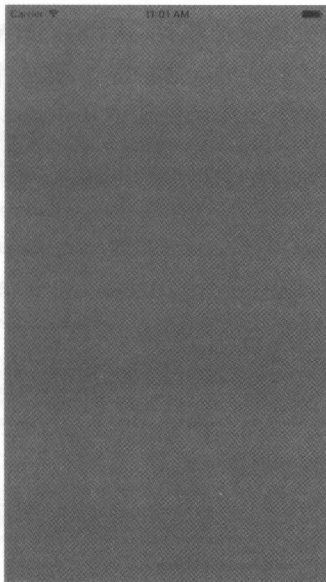


图 2.14 使用 flex 属性实现自适应屏幕的尺寸大小

2.5 如何调试 React Native 项目

在实际开发中，还有一个影响开发效率的重要因素，即调试。

在 1.4.3 节中已经介绍了 Enable Live Debugger 的使用。本节来介绍另一个非常重要的调试选项 Debug JS Remotely 选项。

(1) 晃动设备或使用模拟器上的快捷键（iOS 模拟器快捷键 `command + D`，Android 模拟器快捷键 `command + M`）打开调试选项，效果如图 2.15 所示。

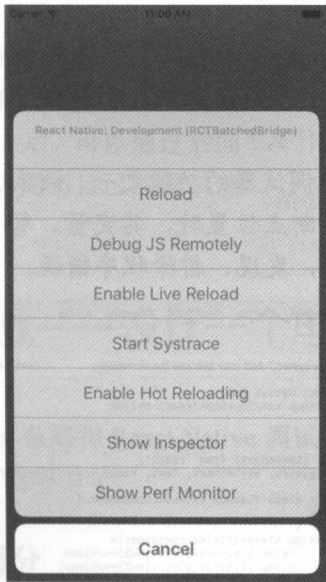


图 2.15 React Native 调试选项

(2) 单击 Debug JS Remotely 选项。此时，React Native 会自动打开 Chrome 浏览器作为调试工具。

(3) 按照如图 2.16 所示的顺序操作，就进入了 React Native 应用的调试状态。

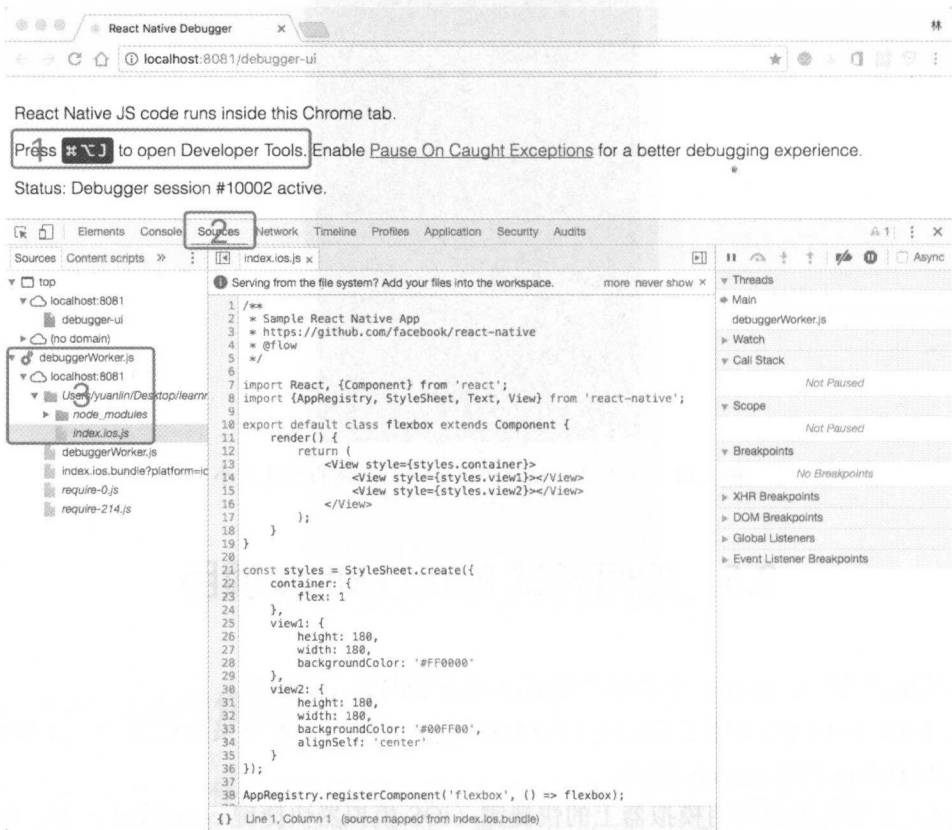


图 2.16 使用 Chrome 浏览器调试 React Native 应用

(4) 在调试状态下，单击 index.ios.js 文件第 12 行的行数来添加一个断点，如图 2.17 所示。

问题：软件开发中的断点是什么？

回答：断点（Breakpoint）是调试器的功能之一，调试时设定断点可以让程序执行到该行程序时停住，借此观察程序到断点位置时，其变量、暂存器、I/O 等相关的变数内容，有助于深入了解程序运作的机制，发现、排除程序错误。

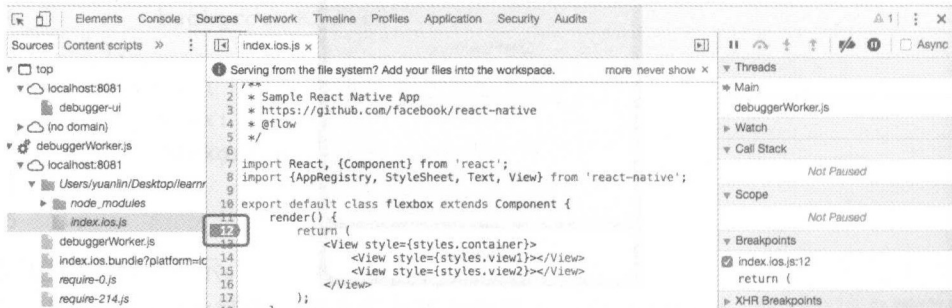


图 2.17 React Native 调试时添加断点

(5) 最后，重新加载运行的应用（iOS 模拟器快捷键 `command + R`，Android 模拟器快捷键 `R + R`）。此时，应用运行到刚才添加断点的第 12 行时就停止了，如图 2.18 所示。

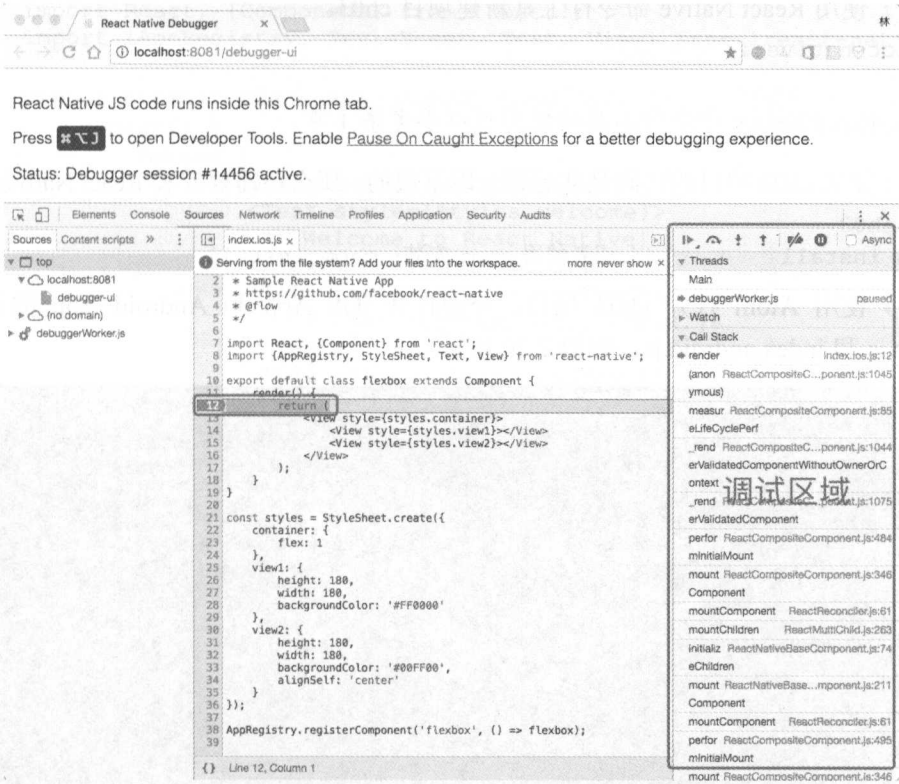


图 2.18 React Native 调试时在断点处暂停运行

此时，可以在右侧的调试区域查看这些信息：当前应用执行的线程状态（Threads）、变量值、调用栈（Call Stack）等信息。而且，还可以使用调试区域上方的指令来实现单步执行、跳过执行、继续执行等调试操作，如图 2.19 所示。

调试技巧和经验是需要 在开发过程中不断积累的，读者在掌握了这些基本用法之后，可以通过后面的例子不断练习，积累开发经验，提高自己的调试能力和开发效率。



图 2.19 React Native 调试时的调试指令

2.6 实战——设计一个电商 App

在掌握了 Git、JSX、Flexbox 布局和 React Native 调试等 React Native 开发的基础知识之后，终于要进入期待已久的实战环节了。


2.6.1 电商 App 的模块划分

实战是不是意味着立刻开始写代码呢？

No, No, No! 在实际的软件开发过程中, 开发并不意味着只有编写代码这一个环节。在编写代码之前, 需要有一个清晰的思路和整体的设计。

(1) 使用 React Native 命令行工具新建项目 ch03。

```
react-native init ch03
```

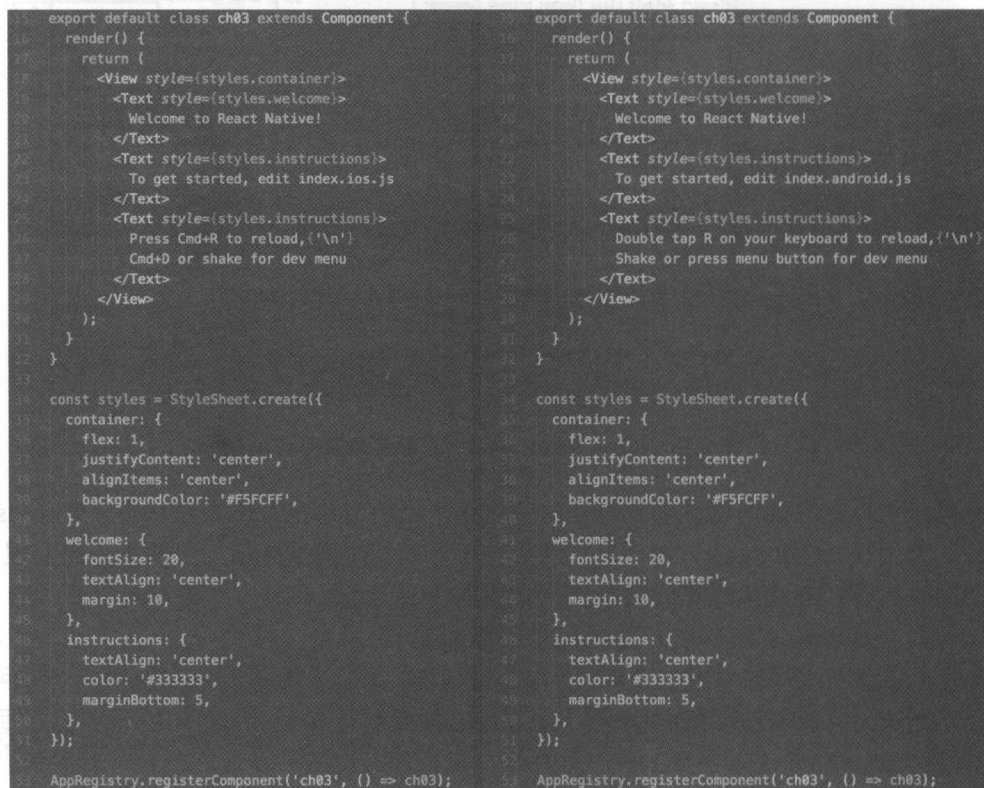
 提示: React Native 命令行工具的使用可以参考第 1 章。

(2) 如果 ch03 项目的代码是从远程仓库下载的, 那么还需要安装 React Native 依赖包。

```
cd ch03
```

```
npm install // 或者使用 cnpm 安装: cnpm install
```

(3) 使用 Atom 打开 ch03 项目。分别打开 iOS App 和 Android App 的入口文件 index.ios.js 和 index.android.js, 如图 2.20 所示。



```

15 export default class ch03 extends Component {
16   render() {
17     return (
18       <View style={styles.container}>
19         <Text style={styles.welcome}>
20           Welcome to React Native!
21         </Text>
22         <Text style={styles.instructions}>
23           To get started, edit index.ios.js
24         </Text>
25         <Text style={styles.instructions}>
26           Press Cmd+R to reload, (\n)
27           Cmd+D or shake for dev menu
28         </Text>
29       </View>
30     );
31   }
32 }
33
34 const styles = StyleSheet.create({
35   container: {
36     flex: 1,
37     justifyContent: 'center',
38     alignItems: 'center',
39     backgroundColor: '#F5FCFF',
40   },
41   welcome: {
42     fontSize: 20,
43     textAlign: 'center',
44     margin: 10,
45   },
46   instructions: {
47     textAlign: 'center',
48     color: '#333333',
49     marginBottom: 5,
50   },
51 });
52
53 AppRegistry.registerComponent('ch03', () => ch03);

```

图 2.20 iOS App 入口文件 index.ios.js 和 Android App 入口文件 index.android.js

 提示: 关于 React Native 项目结构和文件的详细说明, 读者可以参考第 1 章。

细心的读者或许已经发现了问题: index.ios.js 和 index.android.js 文件里的逻辑和内容几乎完全相同。

虽然, 独立的平台入口文件可以让 React Native 开发的 iOS App 和 Android App 拥有自己独立的设计和实现, 但是这样却不能最大限度地发挥 React Native 的优势, 即跨平台应用开发时逻辑和实现的复用。

(4) 为了解决这个问题, 可以将 iOS、Android 平台相同的逻辑和实现放到一个单独的文件中。在 ch03 项目中新建 app.js 文件, 并添加代码如下:

```

01 import React, {Component} from 'react';
02 import {AppRegistry, StyleSheet, Text, View} from 'react-native';
03
04 export default class app extends Component { // 新建名为 app 的组件
05   render() { // 渲染页面的函数
06     return (
07       <View style={styles.container}>
08         <Text style={styles.welcome}>
09           Welcome to React Native!
10         </Text>
11         <Text style={styles.instructions}>
12           To get started, edit index.js
13         </Text>
14         <Text style={styles.instructions}>
15           Press Cmd+R (iOS) or Double tap R (Android) to
16           reload,{'\n'}
17           Shake or press menu button for dev menu
18         </Text>
19       </View>
20     );
21   }
22
23   const styles = StyleSheet.create({ // 创建样式
24     container: { // 页面根 View 的样式
25       flex: 1,
26       justifyContent: 'center',
27       alignItems: 'center',
28       backgroundColor: '#F5FCFF'
29     },
30     welcome: { // “欢迎”文本的样式
31       fontSize: 20,
32       textAlign: 'center',
33       margin: 10
34     },
35     instructions: { // “说明”文本的样式
36       textAlign: 'center',
37       color: '#333333',
38       marginBottom: 5
39     }
40   });

```

app.js 文件中的代码和之前 index.ios.js 和 index.android.js 文件的代码几乎相同, 所以在此就不详细解释了。

(5) 修改 index.ios.js 的代码如下:

```

01 import React, {Component} from 'react';
02 import {AppRegistry} from 'react-native';
03 import app from './app';
04
05 AppRegistry.registerComponent('ch03', () => app);

```

这里需要说明的是, 修改前的 index.ios.js 是直接使用本文件内定义的组件 ch03, 代码如下:

```

01 export default class ch03 extends Component {
02   // 详情参考新建 ch03 项目时的代码

```

```

03 }
04
05 AppRegistry.registerComponent('ch03', () => 'ch03');

```

(6) 现在, 通过 `import` 方式导入刚才新建的组件 `app`。同时, 将 `registerComponent` 方法的第 2 个参数修改为导入的 `app` 组件。

此时, 当重新加载 iOS App 时, 效果到底如何呢? 效果如图 2.21 所示。

(7) 既然 iOS App 运行正常, 那赶紧再看看如何修改 `index.android.js` 文件吧。
`index.android.js` 文件的代码修改如下:

```

01 import React, {Component} from 'react';
02 import {AppRegistry} from 'react-native';
03 import app from './app';
04
05 AppRegistry.registerComponent('ch03', () => app);

```

重新加载 Android App, 效果如图 2.22 所示。

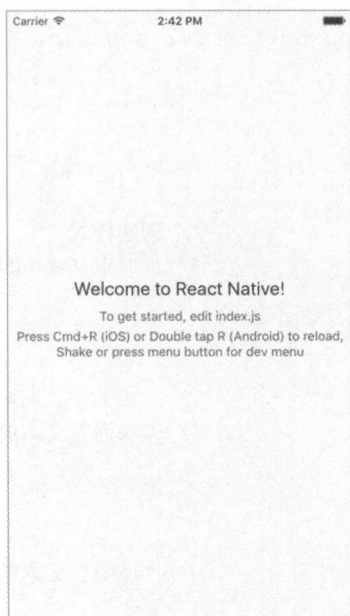


图 2.21 使用 `app.js` 的 iOS App 运行效果

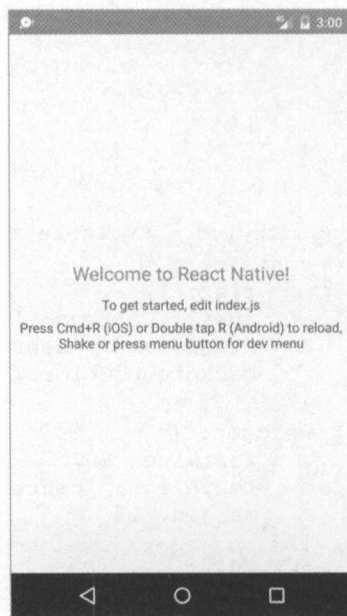



图 2.22 使用 `app.js` 的 Android App 运行效果

 **提示:** 新建 React Native 项目后, 首次执行 `react-native run-android` 可能会发生 `The SDK directory does not exist` 的错误 (如图 2.23 所示), 此时需要在 React Native 项目的 `android` 文件夹下新建 `local.properties` 文件, 并添加配置 `sdk.dir=/path/to/android/sdk/tools`。

这样只需要维护 `app.js` 这一个文件, 就可以实现 iOS App 和 Android App 的同时更新了。最后回顾一下当前 `ch03` 项目的文件结构:

```

index.ios.js // iOS App 入口文件
---- 引用 app.js
index.android.js // Android App 入口文件
---- 引用 app.js

```

```
+ ch03 git:(master) $ react-native run-android
JS server already running.
Building and installing the app on the device (cd android && ./gradlew installDebug)...

FAILURE: Build failed with an exception.

* What went wrong:
A problem occurred configuring project ':app'.
> The SDK directory '/usr/local/opt/android-sdk' does not exist.

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.

BUILD FAILED

Total time: 5.47 secs
Could not install the app on the device, read the error above for details.
Make sure you have an Android emulator running or a device connected and have
set up your Android development environment:
https://facebook.github.io/react-native/docs/android-setup.html
```

图 2.23 react-native run-android 找不到 Android SDK 错误

除了上述不同平台入口复用 `app.js` 的设计之外，完整的软件开发流程还包括需求分析与设计、软件概要与详细设计、编写代码、测试、发布与更新等一系列的流程。就 React Native 开发来说，开发的一般流程如图 2.24 所示。

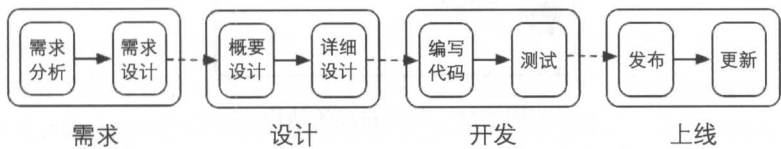


图 2.24 React Native 应用开发的一般流程

提示：上述 React Native 开发流程基于传统的“瀑布流”，虽然“瀑布流”是比较传统的开发流程，但是当今软件开发中流行的“敏捷开发”和“快速迭代”仍然是基于“瀑布流”的，所以了解和熟悉“瀑布流”的软件开发流程仍然很有必要。

本书除了需求环节不做讨论之外，无论是设计、开发（测试暂不包含）以及上线都会有详细的介绍和说明，笔者也非常期望本书能帮助读者叩开 React Native 开发的大门。

2.6.2 设计首页布局

首先笔者对要开发的应用做个简单的了解和框架性的解剖。通常，电商类的应用首页主要由如图 2.25 所示的几个部分组成。

提示：为了实现上述产品设计效果图，本书使用的工具是 Sketch (<https://www.sketchapp.com/>)，它是一款流行的矢量图设计工具，深受移动平台设计师的喜爱。如果读者感兴趣的话，可以下载试用版体验一下。

首页布局的结构从上到下依次如下。

- 状态栏：显示设备网络、时间、电量等信息。
- 搜索框：提供搜索输入框和按钮，用于搜索商品。
- 轮播广告：循环播放广告和推荐信息。

- 商品列表：展示所有商品的列表。

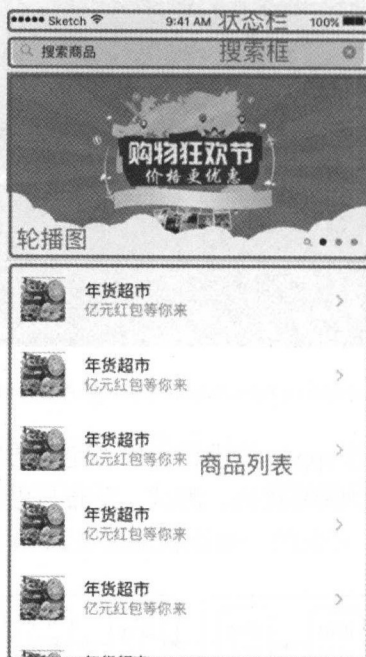


图 2.25 首页布局的结构

按照上述分析的页面结构，修改 app.js 文件的代码如下：

```
01 export default class app extends Component {
02   render() {
03     return (
04       <View style={styles.container}>
05         <View style={styles.searchbar}> // 搜索框
06           <Text>
07             搜索框
08           </Text>
09         </View>
10         <View style={styles.advertisement}> // 轮播广告
11           <Text>
12             轮播广告
13           </Text>
14         </View>
15         <View style={styles.products}> // 商品列表
16           <Text>
17             商品列表
18           </Text>
19         </View>
20       </View>
21     );
22   }
23 }
24
25 const styles = StyleSheet.create({
26   container: {
27     flex: 1
28   },
29   searchBar: { // 搜索框的样式
```

```

30      marginTop: 20, // 空出了状态栏的高度 20
31      height: 40,
32      backgroundColor: 'red',
33      justifyContent: 'center',
34      alignItems: 'center'
35    },
36    advertisement: { // 轮播广告样式
37      height: 180,
38      backgroundColor: 'green',
39      justifyContent: 'center',
40      alignItems: 'center'
41    },
42    products: { // 商品列表样式
43      flex: 1,
44      backgroundColor: 'blue',
45      justifyContent: 'center',
46      alignItems: 'center'
47    }
48  });

```

⚠注意：因为状态栏比较特殊，所以这里没有明确配置状态栏，而使用了默认配置。更多关于状态栏的使用和详细配置，将在 2.6.6 节中详细讨论。

上述代码中 `marginTop` 的用法和 2.4 节中的 CSS “盒子模型”是完全一样的，只是因为 JavaScript 采用“驼峰命名法”，所以才和 CSS “盒子模型”中的 `margin-top` 有所差异。

先重新加载 iOS App 看下效果，如图 2.26 所示。这里用到了一个 Flexbox 布局的技巧。

- 搜索框高度是固定的 40，即 `height: 40`。
- 轮播广告高度是固定的 180，即 `height: 180`。
- 商品列表高度的需求应该是除状态栏、搜索框、轮播广告之外屏幕剩余的高度，解决办法就是 `flex: 1`。

接着再来看一下应用在 Android 设备上的运行效果，如图 2.27 所示。

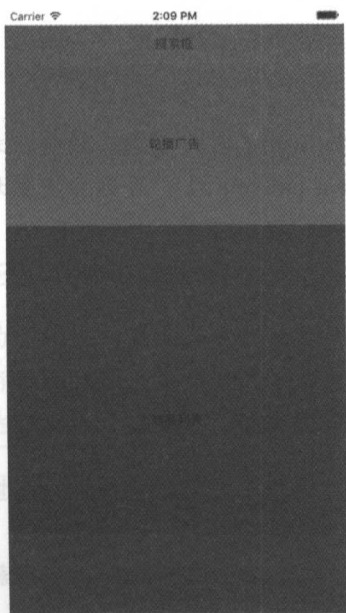


图 2.26 iOS App 首页布局

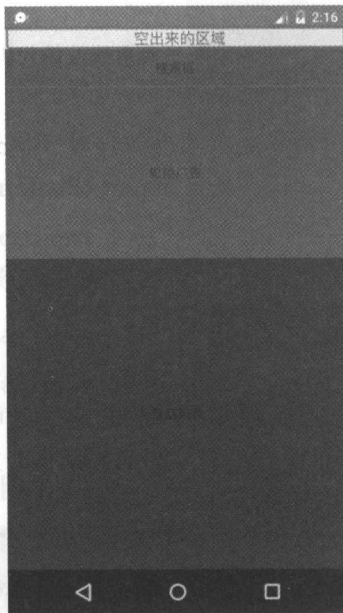


图 2.27 Android App 首页布局

Android App 显示有点小问题：搜索框和状态栏之间空出了一块区域！


为什么在 iOS 上好好的，在 Android 上显示效果却不一样呢？

原来，这是因为 iOS 和 Android 对状态栏处理的方式不同导致的。在默认情况下，iOS 平台内容显示区域是从屏幕顶部开始的，为了不和状态栏重叠，代码中添加了 `marginTop: 20`；而 Android 平台内容显示区域默认就是不包括状态栏的，所以代码中添加了 `marginTop: 20` 就变成了空出来的这块区域。

当然，React Native 考虑到了平台的差异性，已经为开发者提供了 `Platform` 来判断当前运行的系统。`Platform.OS` 在 iOS 系统上会返回 `ios`，而在 Android 设备或模拟器上则会返回 `android`，据此，可以在不同平台上设置不同的 `marginTop` 值。

所以，使用 `Platform` 来修改 `app.js` 文件中的布局代码如下：

```
01 import React, {Component} from 'react';
02 import {
03   AppRegistry,
04   StyleSheet,
05   View,
06   Text,
07   Platform          // 注意：这里必须首先 import Platform
08 } from 'react-native';
09
10 // 这里省略了没有修改的代码
11
12 const styles = StyleSheet.create({
13   // 这里省略了没有修改的代码
14   searchbar: {
15     marginTop: Platform.OS === 'ios'
16       ? 20
17       : 0, // iOS 设备上 marginTop 为 20，Android 设备上 marginTop 为 0
18     height: 40,
19     backgroundColor: 'red',
20     justifyContent: 'center',
21     alignItems: 'center'
22   },
23   // 这里省略了没有修改的代码
24 });
```

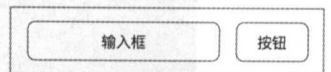
 **注意：**当使用新的模块或组件时（例如这里的 `Platform`）时，首先必须要在文件头导入该模块或组件（`import {Platform} from 'react-native';`），否则会发生无法找到变量 `Platform` 的错误。

2.6.3 实现搜索栏

在理清了应用的整体布局之后，就可以集中精力来依次实现每个 UI 组件。对于顶部的搜索框，主要由以下两个部分组成，如图 2.28 所示。

React Native 已经为开发者提供好了相应的 UI 组件：输入框使用 `TextInput` 组件，按钮使用 `Button` 组件。

所以，修改 `app.js` 文件的代码如下：



搜索框


图 2.28 顶部搜索框的结构

```

01 export default class app extends Component {
02   render() {
03     return (
04       <View style={styles.container}>
05         <View style={styles.searchbar}>
06           <TextInput style={styles.input} placeholder='搜索
            商品'></TextInput>
07           <Button style={styles.button} title='搜索'></Button>
08         </View>
09         <View style={styles.advertisement}>
10           <Text>
11             轮播广告
12           </Text>
13         </View>
14         <View style={styles.products}>
15           <Text>
16             商品列表
17           </Text>
18         </View>
19       </View>
20     );
21   }
22 }
23
24 const styles = StyleSheet.create({
25   // 这里省略了没有修改的代码
26   searchbar: {
27     marginTop: Platform.OS === 'ios'
28       ? 20
29       : 0,
30     height: 40,
31     flexDirection: 'row'
32   },
33   input: {
34     flex: 1,
35     borderColor: 'gray',
36     borderWidth: 2
37   },
38   button: {
39     flex: 1
40   },
41   // 这里省略了没有修改的代码
42 });

```

其中, `TextInput` 中 `Placeholder` 的作用是在 `TextInput` 中还没有输入任何文字时, 显示提示语。运行效果如图 2.29 所示。

 **提示:** 为了描述的简洁, 这里没有将多个平台的运行效果全部截图说明。本书如果没有特别说明的话, 默认 iOS 和 Android 的效果是相同的。

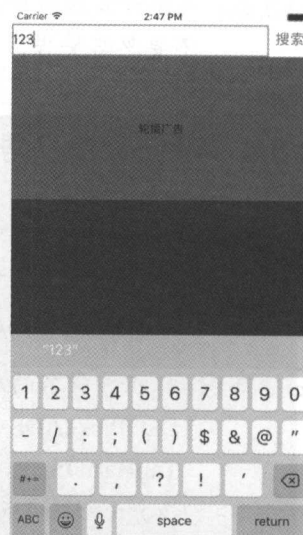


图 2.29 搜索框

不过细心的读者可能会发现这样一个问题, 如图 2.30 所示。

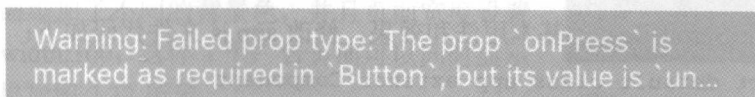


图 2.30 React Native 警告

单击这个警告，查看警告的详细描述，如图 2.31 所示。

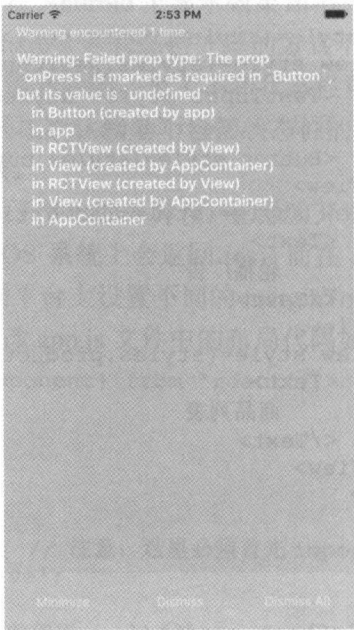




图 2.31 查看警告的详细描述

这个警告是告诉开发者：Button 的 onPress 属性没有实现。由于按钮的响应事件处理会在 2.6.6 节中详细讨论，所以这里选择 Dismiss 或 Dismiss All 忽略该警告即可。

 **注意：**实际开发过程中，开发者需要仔细阅读警告信息，并解决警告描述的问题，千万不要忽略警告！

2.6.4 设计轮播广告

React Native 提供的 ScrollView 和 ViewPager 组件都可以实现轮播广告的效果。但是 ViewPager 是 Android 平台特有的组件，因此，为了考虑平台兼容性和代码复用性，这里使用 ScrollView 来实现轮播效果。

 **小知识：**从组件命名的后缀可以看出哪些组件是 iOS 平台特有的，例如，DatePickerIOS、PickerIOS、ProgressViewIOS、SegmentedControlIOS 以及 TabBarIOS 等，哪些组件是 Android 平台特有的，例如，DrawerLayoutAndroid、ProgressBarAndroid、ToolbarAndroid 以及 ViewPagerAndroid 等。

1. ScrollView的使用

首先来看下 ScrollView 的基本用法和效果，修改 app.js 文件代码如下：

```
01 export default class app extends Component {
02   render() {
03     return (
```

```

04      <View style={styles.container}>
05          // 这里省略了没有修改的代码
06          <View style={styles.advertisement}>
07              <ScrollView
08                  ref="scrollView"      // 可以使用 this.refs.
09                                      scrollView 来获取该组件
10                  horizontal={true}      // 横向滚动
11                  showsHorizontalScrollIndicator={false} // 不显示横向滚动条
12                  pagingEnabled={true}    // 分页
13              ><Text style={{
14                  width: 100,
15                  height: 180,
16                  backgroundColor: 'gray'
17              }}>广告 1</Text>
18              <Text style={{
19                  width: 100,
20                  height: 180,
21                  backgroundColor: 'orange'
22              }}>广告 2</Text>
23              <Text style={{
24                  width: 100,
25                  height: 180,
26                  backgroundColor: 'yellow'
27              }}>广告 3</Text>
28          </ScrollView>
29      </View>
30      // 这里省略了没有修改的代码
31  </View>
32  );
33  }
34
35  const styles = StyleSheet.create({
36      // 这里省略了没有修改的代码
37      advertisement: {
38          height: 180
39      },
40      // 这里省略了没有修改的代码
41  });

```

ScrollView 的滚动方向默认是纵向的，为了实现横向轮播图的效果，将其滚动方向设置成水平，即 `horizontal={true}`，同时还开启了分页效果 `pagingEnabled={true}`。另外，在 ScrollView 中，添加了 3 个 Text 子组件，分别显示 3 个广告页面。这里为了更清楚地看到所有页面，给 3 个广告页面添加了不同的背景色，并且页面宽度设置得比较小，如 `width:100`，这样 3 个广告页面可以同时显示在屏幕上。效果如图 2.32 所示。

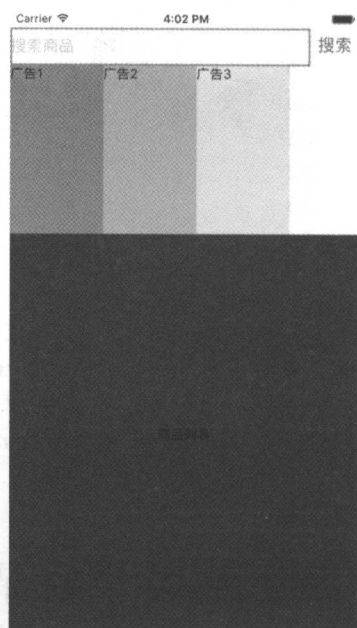


图 2.32 ScrollView 实现轮播图

2. 完善轮播广告

在熟悉了 `ScrollView` 的基本用法之后，接着来进一步完善轮播广告的效果：设置每一个广告页的宽度为屏幕宽度。

同样，React Native 已经提供了 API: `Dimensions` 来获取屏幕的宽高。

 **提示：**更多关于 React Native API 的内容，本书后面的章节会有详细和完整的说明。

所以，将 `Text` 组件的宽度从 `width: 100` 修改成 `width: Dimensions.get('window').width`，详细代码如下：

```

01 export default class app extends Component {
02     render() {
03         return (
04             <View style={styles.container}>
05                 // 这里省略了没有修改的代码
06                 <View style={styles.advertisement}>
07                     <ScrollView
08                         ref="scrollView"
09                         horizontal={true}
10                         showsHorizontalScrollIndicator={false}
11                         pagingEnabled={true}
12                     ><Text style={{
13                         width: Dimensions.get('window').width,
14                         height: 180,
15                         backgroundColor: 'gray'
16                     }}>广告 1</Text>
17                     <Text style={{
18                         width: Dimensions.get('window').width,
19                         height: 180,
20                         backgroundColor: 'orange'
21                     }}>广告 2</Text>
22                     <Text style={{
23                         width: Dimensions.get('window').width,
24                         height: 180,
25                         backgroundColor: 'yellow'
26                     }}>广告 3</Text>
27                 </ScrollView>
28             </View>
29             // 这里省略了没有修改的代码
30         </View>
31     );
32 }
33 }
```

重新加载应用，效果如图 2.33 所示。

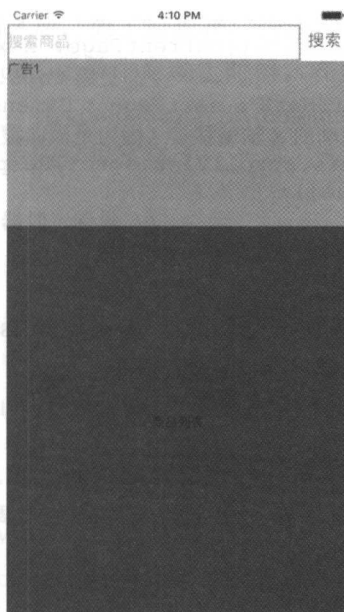


图 2.33 改进的轮播图效果

3. 让轮播广告动起来

距离完整的轮播图效果就只差一步了：让轮播图动起来，即每隔一段时间自动切换到下一页面，并且当处在最后一个页面时，再回滚到第一个页面。


在 `app.js` 文件中添加代码如下：

```
01 export default class app extends Component {
02   constructor(props) {           // 构造函数
03     super(props);
04     this.state = {
05       currentPage: 0
06     };
07   }
08
09   render() {
10     // 这里省略了没有修改的代码
11   }
12
13   componentDidMount() {
14     this._startTimer();
15   }
16
17   componentWillUnmount() {
18     clearInterval(this.interval);
19   }
20
21   _startTimer() {
22     this.interval = setInterval(() => {    // 使用 setInterval() 创
                                           // 建定时器
23       nextPage = this.state.currentPage + 1;
24       if (nextPage >= 3) {
25         nextPage = 0;    // 如果已经滚动到最后一页，下次返回第一页
26       }
27     }, 3000);
28   }
29 }
```



```
27
28         this.setState({currentPage: nextPage});
                                   // 更新 this.state 中 currentPage 的值
29         const offsetX = nextPage * Dimensions.get('window').width;
// 计算 ScrollView 滚动的 x 轴偏移量 (因为是横向滚动)
30         this.refs.scrollView.scrollResponderScrollTo({x: offsetX,
  y: 0, animated: true});
31     }, 2000); // 设置定时器的间隔为 2s
32   }
33 }
```

在上述代码中，读者需要关注的是构造函数中的 `this.state`。从代码中可以看出，`state` 是一个字典类型的数据，主要用于存储组件相关的数据，例如，轮播图当前显示页面的序号。

 **提示：**除了 `state`，`props` 也可以用来存储组件相关的数据。但是 `props` 通常是在父组件中指定的，而且一经指定，在被指定的组件的生命周期中则不再改变，所以对于需要改变的数据，使用 `state`。

4. 组件的生命周期

上述实现中，可能会让读者感到困惑的是 `componentDidMount()` 和 `componentWillUnmount()` 函数。这里来深入了解下 React Native 开发中组件的生命周期以及相关的函数（包括 `componentDidMount()` 和 `componentWillUnmount()`）。

在 React Native 项目中，所有展示的界面都可以看做是一个组件（Component），只是功能和逻辑的复杂程度不同。例如，`app.js` 文件中声明的 `app` 组件，它就是继承自 `React.Component`，代码如下：

```
01 import React, {Component} from 'react';
02 import {AppRegistry, StyleSheet, Text, View} from 'react-native';
03
04 export default class flexbox extends Component {
05   // 这里省略了无关的代码
```

组件的生命周期可以分为加载、更新以及卸载，每一个生命周期都提供了一些方法供开发者重写，以实现相应的需求和功能。其中，最常用的加载和卸载的函数调用流程如图 2.34 所示。

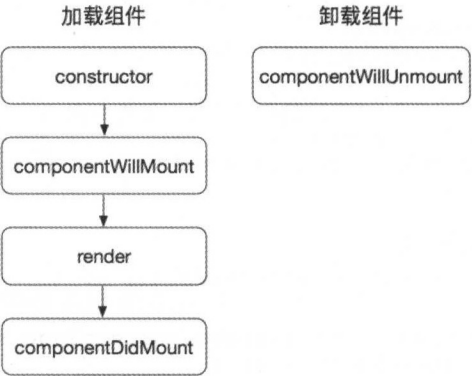


图 2.34 组件生命周期

在了解了组件的生命周期之后，为了给轮播广告添加自动滚动的效果，可以在页面渲染好之后的 `componentDidMount()` 函数中启动定时器，在定时器中按照一定的时间间隔自动播放下一页；当组件卸载时，在 `componentWillUnmount()` 函数中销毁定时器。

至此，一个满意的轮播广告效果也实现了。

2.6.5 展示商品列表

列表功能是应用开发中最常见的功能之一，为了实现商品列表，可以使用 React Native 提供的 `ListView` 组件。

`ListView` 组件是 React Native 开发中常用的组件，同时也是 React Native 最核心的组件之一，主要用来高效地显示一个可以垂直滚动变化的数据列表。对于如此重要的组件，赶紧来一探究竟吧。

在 `app.js` 文件中添加 `ListView` 相关代码如下：

```
01 const ds = new ListView.DataSource({      // 创建 ListView.DataSource 数据源
02   rowHasChanged: (r1, r2) => r1 !== r2
03 });
04
05 export default class app extends Component {
06   constructor(props) {
07     super(props);
08     this.state = {
09       currentPage: 0,
10       dataSource: ds.cloneWithRows([      // 为数据源传递一个数组
11         '商品 1',
12         '商品 2',
13         '商品 3',
14         '商品 4',
15         '商品 5',
16         '商品 6',
17         '商品 7',
18         '商品 8',
19         '商品 9',
20         '商品 10'
21       ])
22     };
23   }
24
25   render() {
26     return (
27       <View style={styles.container}>
28         // 这里省略了没有修改的代码
29         <View style={styles.products}>
30           <ListView dataSource={this.state.dataSource} renderRow=
31             {this._renderRow}/>
32         </View>
33       </View>
34     );
35   }
36   // 这里省略了没有修改的代码
```

```

37
38   _renderRow = (rowData, sectionID, rowID) => {
39     return (
40       <View style={styles.row}>
41         <Text>{rowData}</Text>    // 标签中使用{value}的方法取值
42       </View>
43     );
44   }
45 }

```

同时，修改样式和布局的代码如下：

```

01 const styles = StyleSheet.create({
02   // 这里省略了没有修改的代码
03   products: {
04     flex: 1
05   },
06   row: {
07     height: 60,
08     justifyContent: 'center',
09     alignItems: 'center'
10   }
11 });

```

此时，ListView 组件的实际运行效果如图 2.35 所示。

从上述例子可以看出，使用 ListView 组件必须实现以下两个属性。

- **dataSource**: ListView 组件的数据源。这里使用 state 来保存数据。提供给数据源的 `rowHasChanged()` 函数告诉 ListView 组件是否需要重绘某一行，即该行数据发生变化时对该行进行重绘。
- **renderRow()**: 该函数根据数据源中每一条数据，返回列表每一行显示的组件。它的函数原型为 `(rowData, sectionID, rowID, highlightRow) => renderable`。

当然，ListView 组件的属性和用法远不止现在介绍的这些，随着应用开发的深入，后面还将介绍更多 ListView 的属性和用法。

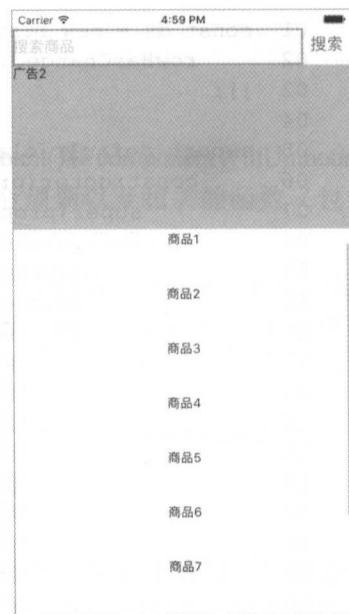


图 2.35 ListView 实现的商品列表

2.6.6 实现交互功能和状态栏

至此，首页的主要内容就已经基本完成了。不知道细心的读者有没有发现这样的问题：本例应用（除搜索框外）还不能接收和响应任何的用户操作。那么现在就来给这些 UI 组件添加单击操作的响应。

1. 让搜索框能响应用户操作

给搜索框的“搜索”按钮添加单击响应，修改搜索框的代码如下：

```

01 export default class app extends Component {
02   // 这里省略了没有修改的代码

```

```

03
04   render() {
05     return (
06       <View style={styles.container}>
07         <View style={styles.searchbar}>
08           <TextInput style={styles.input} placeholder='搜索
           商品'></TextInput>
09           <Button
10             style={styles.button}
11             title='搜索'
12             onPress={() => Alert.alert('你单击了搜索按钮',
              null, null)}></Button>
13         </View>
14         // 这里省略了没有修改的代码
15       </View>
16     );
17   }
18 }

```

从上述代码中可以看出，为 Button 组件添加单击响应比较简单，只需要实现 onPress() 函数。在 onPress() 函数中，使用了一个新的 Alert 组件用来弹出提醒，提示用户单击了搜索按钮，如图 2.36 所示。

提示：实现 Button 组件的 onPress() 函数不仅仅可以添加单击响应，还解决了 2.6.3 节中顶部搜索栏目中的警告问题。

2. 给轮播图和商品列表添加单击响应

这里读者可能会犯难：轮播图和商品列表不是 Button 组件（轮播图是 Text 组件，商品列表是 View 组件），所以需要使用一个新的 React Native 组件 TouchableHighlight。

TouchableHighlight 组件主要用于封装其他组件，使其可以正确响应单击操作，代码如下：

```

01 export default class app extends Component {
02   // 这里省略了没有修改的代码
03
04   render() {
05     return (
06       <View style={styles.container}>
07         // 这里省略了没有修改的代码
08         <View style={styles.advertisement}>
09           <ScrollView ref="scrollView"
10             horizontal={true}
11             showsHorizontalScrollIndicator={false}
12             pagingEnabled={true}>
13             <TouchableHighlight onPress={() => Alert.alert
              ('你单击了轮播图', null, null)}>
14               <Text
15                 style={{
16                   width: Dimensions.get('window').width,
17                   height: 180,

```

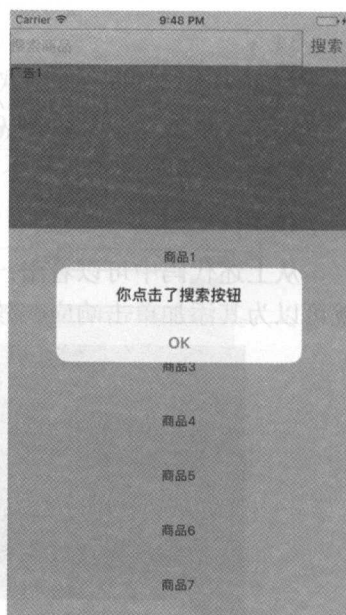


图 2.36 搜索按钮单击响应

```

18         backgroundColor: 'gray'
19       }}>广告 1</Text>
20     </TouchableHighlight>
21     // 这里省略了和上面相似的代码
22   </ScrollView>
23 </View>
24 // 这里省略了没有修改的代码
25 </View>
26   );
27 }
28
29 // 这里省略了没有修改的代码
30
31 _renderRow = (rowData, sectionID, rowID) => {
32   return (
33     <TouchableHighlight onPress={() => Alert.alert('你单击了商
      品列表', null, null)}>
34       <View style={styles.row}>
35         <Text>{rowData}</Text>
36       </View>
37     </TouchableHighlight>
38   );
39 }
40 }

```

从上述代码中可以看出：把想要添加单击响应的组件放到 `TouchableHighlight` 组件中，就可以为其添加单击响应，效果如图 2.37 和图 2.38 所示。

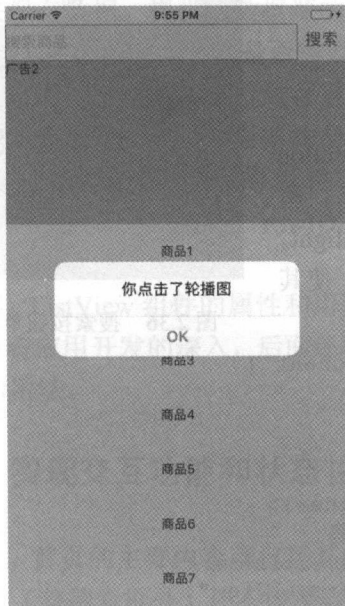


图 2.37 轮播广告的单击响应

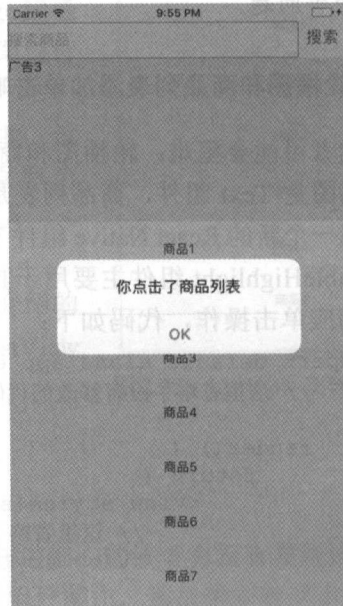


图 2.38 商品列表的单击响应

3. 实现状态栏

还有一个差点被遗忘的组件：状态栏 `StatusBar`。上面的例子使用的是 `StatusBar` 的默认配置和效果，不仅如此，也可以根据自己的需求来配置它，代码如下：

```

01 export default class app extends Component {
02   // 这里省略了没有修改的代码
03
04   render() {
05     return (
06       <View style={styles.container}>
07         <StatusBar backgroundColor={'blue'} // 设置背景色为蓝色
08         barStyle={'default'} // 设置默认样式
09         networkActivityIndicatorVisible={true} // 显示正在请求网络的状态
10       ></StatusBar>
11       // 这里省略了没有修改的代码
12     </View>
13   );
14 }
15
16 // 这里省略了没有修改的代码
17 }

```

上述代码将 `StatusBar` 的背景色设置成了蓝色、样式设置为默认，同时，让状态栏显示正在进行网络请求的状态。运行 Android App 的效果如图 2.39 所示。

背景色修改成功，但是好像并没有显示网络请求的状态，不急，再运行 iOS App 看看，发现效果也有一些不同：背景颜色没有生效，如图 2.40 所示。

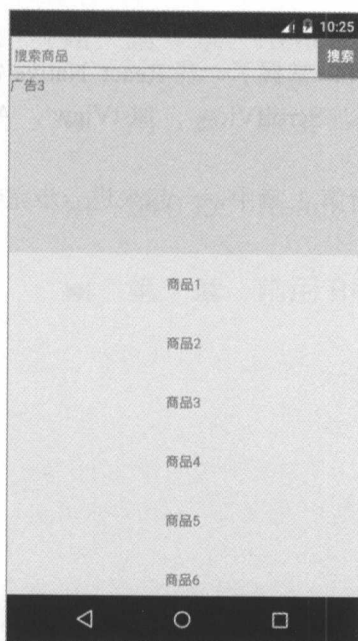


图 2.39 Android App 状态栏

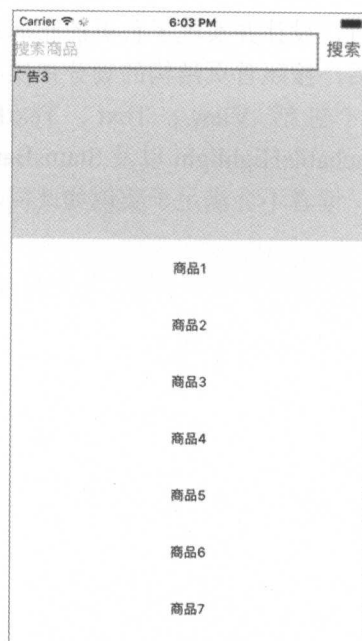


图 2.40 iOS App 状态栏

为什么会这样呢？原来，由于 iOS 和 Android 平台状态栏的差异性，React Native 的 `StatusBar` 组件的部分属性只在特定平台生效。

例如，`StatusBar` 组件常用属性的平台兼容性如下。

- `animated` 和 `hidden` 属性在 iOS、Android 平台都有效。
- `backgroundColor` 和 `translucent` 属性只在 Android 平台有效。

- `barStyle`、`networkActivityIndicatorVisible` 以及 `showHideTransition` 属性只在 iOS 平台有效。


所以就有了上面的平台差异性。

2.7 小 结

不知不觉中，已经完成了电商类应用首页 UI 的全部内容，不知道读者是不是越来越喜欢 React Native 开发了昵？

温故而知新，在开始下一步“征程”之前，先来简单梳理下本章学习到的 React Native 知识。

- 前期，我们进行了充分的准备工作：了解了 Git 的使用、JSX 语法、Flexbox 布局以及 React Native 调试的方法。
- 然后，重新设计了应用的入口，让 iOS App 和 Android App 复用相同的组件 `app.js`，并且了解了 React Native 应用开发的一般流程。

 **提示：**这里分享给读者的经验是，先搭好架子是一个良好的开发习惯。

- 接着，设计并实现了首页的布局：使用 Flexbox 布局。
- 最后，按照首页结构的划分依次实现了各个组件，掌握了一些 React Native 常用组件（包括 `View`、`Text`、`TextInput`、`Button`、`ScrollView`、`ListView`、`Alert`、`TouchableHighlight` 以及 `StatusBar` 等）的用法。

当然，读者不会满足于实现如此简单的 UI 效果，第 3 章中，一起来进一步完善我们的应用吧。

第2篇

React Native 应用开发实战

- » 第3章 React Native 的组件（1）
- » 第4章 React Native 的组件（2）
- » 第5章 原生平台的适配和调试
- » 第6章 React Native 的服务器端处理
- » 第7章 常用 React Native API

第3章 React Native 的组件（1）

第2章中已经实现了电商应用的首页，虽然界面和功能看起来略简陋了点，但是麻雀虽小，五脏俱全，首页的整体布局和大概效果已经完整地呈现在我们面前了。接下来，在“夯实的地基”上，我们将继续“盖楼”，为应用实现更多的功能和更好的体验。

本章主要内容有：

- 移植旧的 App 项目。
- 学会重构现有代码。
- 完善第2章中的搜索框。
- 完善第2章中的轮播广告。
- 完善第2章中的商品列表。
- 实现页面跳转。
- 实现页面间的数据传递。


3.1 创建新的电商 App

之前创建了一个简单的电商项目，本节来实现对该项目的重构。

3.1.1 移植旧电商项目

（1）先创建 React Native 项目并安装依赖包。

```
react-native init ch04      // 新建 React Native 项目 ch04
cd ch04
npm install                 // 或者使用 cnpm 安装: cnpm install
```

 **小知识：** npm install 命令还可以简写成 npm i，更多说明可以使用 npm help install 查看帮助文档。

（2）将第2章 ch03 项目中的 index.ios.js、index.android.js 以及 app.js 文件都复制到 ch04 文件夹中。完成文件复制和覆盖之后，ch04 项目的结构如图 3.1 所示。

但是，如果这时候直接运行 App，会发生应用 ch04 没有注册的错误，如图 3.2 所示。

发生错误的 AppRegistry.registerComponent 函数在第1章的例子中就遇到过，那么它到底有什么作用呢？

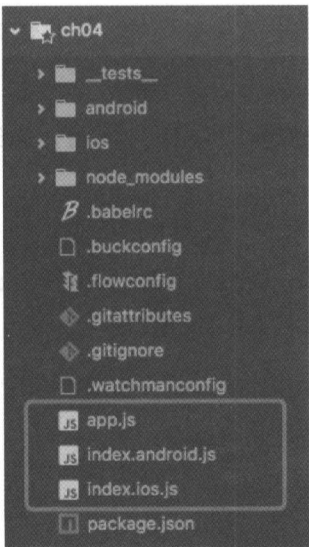


图 3.1 ch04 项目的结构

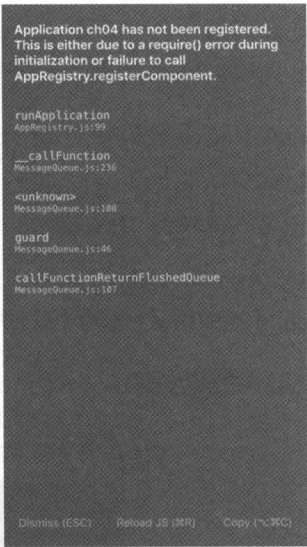


图 3.2 应用 ch04 没有注册的错误

原来，AppRegistry 是所有 React Native 应用的入口，应用的根组件通过 AppRegistry.registerComponent 方法注册自己，然后原生系统才可以加载 React Native 应用的代码并运行 React Native 应用。流程如图 3.3 所示。

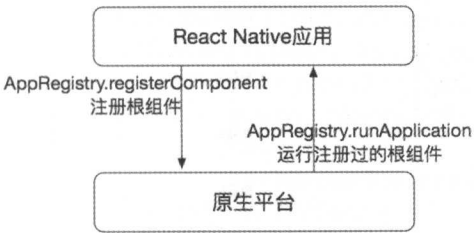


图 3.3 AppRegistry.registerComponent 注册 React Native 应用的根组件

(3) 了解 AppRegistry 的原理之后，就可以轻松地找到错误的原因，index.ios.js 和 index.android.js 文件中注册的应用名称不正确，React Native 应用中注册的应用是 ch03，而 react-native 命令行工具生成的原生代码运行的应用是 ch04，产生错误的代码如下：

```
// index.ios.js 和 index.android.js 文件
AppRegistry.registerComponent('ch03', () => app); // React Native 注册 ch03

// iOS 原生项目中的 AppDelegate.m 文件
RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:jsCodeLocation
                                                                    moduleName:@"ch04" // 原生代码运行 ch04
                                                                    initialProperties:nil
                                                                    launchOptions:launchOptions];

// Android 原生项目中的 MainActivity.java 文件
@Override
protected String getMainComponentName() {
    return "ch04"; // 原生代码运行 ch04
}
```

🔔提示：这里是我们第一次接触到 React Native 开发中的原生代码，通常情况下不需要修改原生代码，只要简单了解即可。

(4) 将 React Native 代码中注册的应用名称 ch03 修改为 ch04 即可，修改 index.ios.js 和 index.android.js 代码如下：

```
01 AppRegistry.registerComponent('ch04', () => app);
```

🔔提示：由于 iOS App 和 Android App 的入口文件不同，分别为 index.ios.js 和 index.android.js，所以 ch04 项目中的这两个文件都需要按照上述代码进行修改。

重新加载应用，效果如图 3.4 所示。

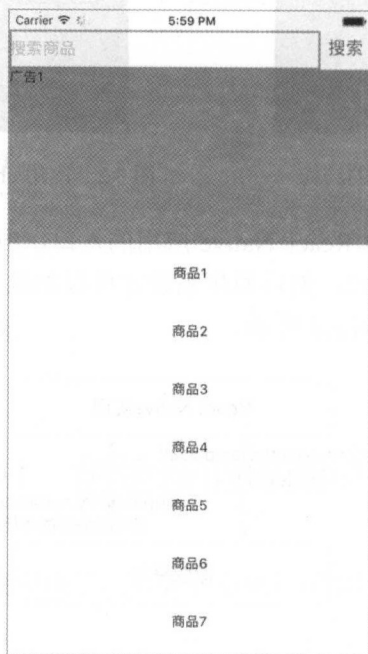


图 3.4 注册正确的应用名称后的运行效果

3.1.2 重构现有的代码

在成功移植 ch03 项目的实现到新建的 ch04 项目后，我们就可以开动了：完善应用。但是，笔者在实际开发过程中总结的经验是，完善的第一步往往是从梳理和重构现有代码开始，而非立即添加新代码或功能。只有不断地重构然后添加新功能，代码的可维护性才会越好，这也是应用稳定性和扩展性的重要保证。

重新审视 app.js 文件的代码，从中不难发现这样的问题：ScrollView 组件下所有子组件的样式都是类似的，导致很多冗余代码。ScrollView 组件现有代码如下：

```
01 export default class app extends Component {  
02     // 这里省略了没有修改的代码  
03  
04     render() {
```

```

05     return (
06         <View style={styles.container}>
07             // 这里省略了没有修改的代码
08             <View style={styles.advertisement}>
09                 <ScrollView ref="scrollView"
10                     horizontal={true}
11                     showsHorizontalScrollIndicator={false}
12                     pagingEnabled={true}>
13                     <TouchableHighlight onPress={() => Alert.alert
14                         ('你单击了轮播图', null, null)}>
15                         <Text style={{
16                             width: Dimensions.get('window').width,
17                             height: 180,
18                             backgroundColor: 'gray'
19                         }}>广告 1</Text>
20                     </TouchableHighlight>
21                     <TouchableHighlight onPress={() => Alert.alert
22                         ('你单击了轮播图', null, null)}>
23                         <Text style={{
24                             width: Dimensions.get('window').width,
25                             height: 180,
26                             backgroundColor: 'orange'
27                         }}>广告 2</Text>
28                     </TouchableHighlight>
29                     <TouchableHighlight onPress={() => Alert.alert
30                         ('你单击了轮播图', null, null)}>
31                         <Text style={{
32                             width: Dimensions.get('window').width,
33                             height: 180,
34                             backgroundColor: 'yellow'
35                         }}>广告 3</Text>
36                     </TouchableHighlight>
37                 </ScrollView>
38             </View>
39             // 这里省略了没有修改的代码
40         </View>
41     );
42 }

```

// 这里省略了没有修改的代码

如果想更新轮播广告的高度的话,就需要修改多处代码 `height: 180`。因此,可以尝试将重复的样式代码抽离出来,代码效果如下:

```

01 export default class app extends Component {
02     // 这里省略了没有修改的代码
03
04     render() {
05         return (
06             <View style={styles.container}>
07                 // 这里省略了没有修改的代码
08                 <View style={styles.advertisement}>

```

```

09         <ScrollView ref="scrollView"
10             horizontal={true}
11             showsHorizontalScrollIndicator={false}
12             pagingEnabled={true}>
13             <TouchableHighlight onPress={() => Alert.alert
14                 ('你单击了轮播图', null, null)}>
15                 <Text style={[
16                     styles.advertisementContent, {
17                         backgroundColor: 'gray'
18                     }
19                 ]}>广告 1</Text>
20             </TouchableHighlight>
21             <TouchableHighlight onPress={() => Alert.alert
22                 ('你单击了轮播图', null, null)}>
23                 <Text style={[
24                     styles.advertisementContent, {
25                         backgroundColor: 'orange'
26                     }
27                 ]}>广告 2</Text>
28             </TouchableHighlight>
29             <TouchableHighlight onPress={() => Alert.alert
30                 ('你单击了轮播图', null, null)}>
31                 <Text style={[
32                     styles.advertisementContent, {
33                         backgroundColor: 'yellow'
34                     }
35                 ]}>广告 3</Text>
36             </TouchableHighlight>
37         </ScrollView>
38     </View>
39     // 这里省略了没有修改的代码
40 </View>
41 );
42 }
43 // 这里省略了没有修改的代码
44 }

```

其中，添加的样式 `advertisementContent` 定义如下：

```

01 const styles = StyleSheet.create({
02     // 这里省略了没有修改的代码
03     advertisementContent: {
04         width: Dimensions.get('window').width,
05         height: 180
06     },
07     // 这里省略了没有修改的代码
08 });

```

这样，当要修改轮播广告页面的高度时，只需要修改样式 `advertisementContent` 这一处，所有广告页面的样式都可以同时更新。

虽然，样式代码做到了复用，但是 `TouchableHighlight` 组件和单击事件等还是重复的，对此，可以使用 JavaScript 数组的 `map()` 方法来做进一步优化，修改 `app.js` 代码如下：

```

01 export default class app extends Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       advertisements: [           // 轮播广告数组
06         { // 数组中的每个成员描述了广告详细信息
07           title: '广告 1',
08           backgroundColor: 'gray'
09         }, {
10           title: '广告 2',
11           backgroundColor: 'orange'
12         }, {
13           title: '广告 3',
14           backgroundColor: 'yellow'
15         }
16       ],
17     };
18   }
19
20   // 这里省略了没有修改的代码
21
22   render() {
23     return (
24       <View style={styles.container}>
25         // 这里省略了没有修改的代码
26         <View style={styles.advertisement}>
27           <ScrollView ref="scrollView"
28             horizontal={true}
29             showsHorizontalScrollIndicator={false}
30             pagingEnabled={true}>
31             {this.state.advertisements.map((advertisement,
32               index) => {
33               return (
34                 <TouchableHighlight key={index} onPress=
35                   {() => Alert.alert('你单击了轮播图', null,
36                     null)}>
37                   <Text style={[
38                     styles.advertisementContent, {
39                       backgroundColor: advertisement.
40                         backgroundColor
41                     ]}>{advertisement.title}</Text>
42                 </TouchableHighlight>
43               );
44             }
45             </ScrollView>
46           </View>
47           // 这里省略了没有修改的代码
48         </View>
49       // 这里省略了没有修改的代码
50     );
51   }

```

此时，如果想要修改、添加或删除广告页，对于重构后的代码就非常简单，只需要编辑 `this.state.advertisements` 数组即可。重新加载应用，效果如图 3.5 所示。

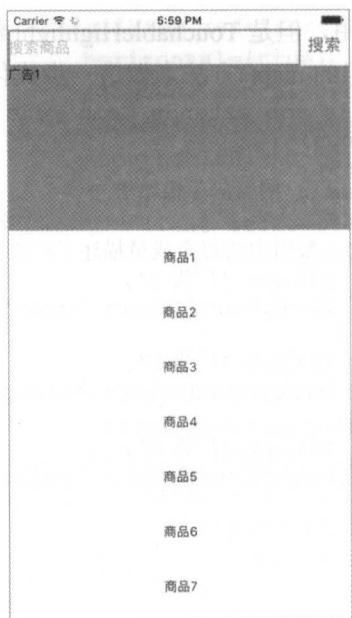
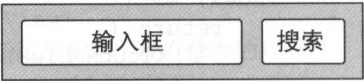


图 3.5 代码重构后应用运行效果

3.2 完善搜索框功能——TextInput 组件

重构代码完毕后，就可以“轻装上阵”，更快更好地为应用添加新功能了。按照之前首页结构的划分，先来看看搜索框。搜索框分为输入框和搜索按钮两部分，如图 3.6 所示。



搜索框

图 3.6 搜索框的结构

用户在输入框输入要搜索的关键字后，单击“搜索”按钮，即可按照输入框中的关键字进行搜索。

3.2.1 搜索提示框

为了实现这样的效果，可以使用 `TextInput` 组件的 `onChangeText()` 方法。当输入框内容变化时会调用此回调函数，改变后的文本内容作为参数传递，然后使用 `this.state.searchText` 保存此时的输入结果。修改 `app.js` 代码如下：

```
01 export default class app extends Component {
02   constructor(props) {
03     super(props);
04     this.state = {
```

```

05         searchText: '', // 保存当前输入的文本
06         // 这里省略了没有修改的代码
07     };
08 }
09
10 // 这里省略了没有修改的代码
11
12 render() {
13     return (
14         <View style={styles.container}>
15             // 这里省略了没有修改的代码
16             <View style={styles.searchbar}>
17                 <TextInput style={styles.input} placeholder='搜索
18                 商品'
19                 onChangeText={({text}) => {
20                     this.setState({searchText: text});
21                 }}></TextInput>
22                 <Button style={styles.button} title='搜索' onPress=
23                 {() => {
24                     Alert.alert('搜索内容 ' + this.state.searchText,
25                         null, null);
26                 }}></Button>
27             </View>
28             // 这里省略了没有修改的代码
29         </View>
30     );
31 }
32
33 // 这里省略了没有修改的代码
34 }

```

重新加载应用，输入要搜索的内容，例如 **Abc**，单击“搜索”按钮，此时提示框将显示输入框刚才输入的 **Abc**，效果如图 3.7 所示。

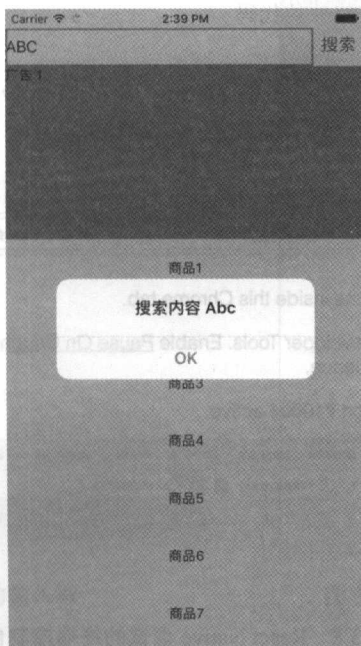


图 3.7 完善搜索框的功能

3.2.2 调试搜索结果

除了使用提示框查看搜索内容的方法之外，这里再介绍另一种高效的调试方法，即使用 `console.log` 将日志打印到终端控制台上。修改 `app.js` 代码如下：

```

01 export default class app extends Component {
02   // 这里省略了没有修改的代码
03
04   render() {
05     return (
06       <View style={styles.container}>
07         // 这里省略了没有修改的代码
08         <View style={styles.searchbar}>
09           <TextInput style={styles.input} placeholder='搜索
            商品'
            onChangeText={(text) => {
10             this.setState({searchText: text});
11             console.log('输入的内容是 ' + this.state.
12               searchText);
13           }}></TextInput>
14           <Button style={styles.button} title='搜索' onPress=
            (() => {
15             Alert.alert('搜索内容 ' + this.state.searchText,
              null, null);
16           }}></Button>
17         </View>
18         // 这里省略了没有修改的代码
19       </View>
20     );
21   }
22
23   // 这里省略了没有修改的代码
24 }

```

然后打开 React Native 调试选项中的 Debug JS Remotely 选项，选择 Chrome 浏览器中的 Console，效果如图 3.8 所示。

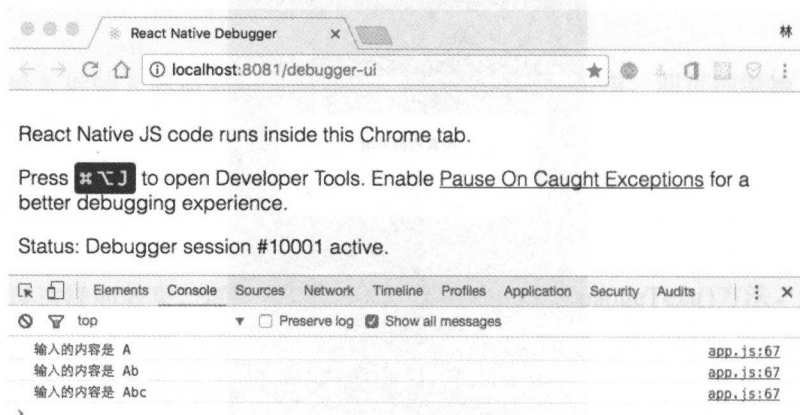


图 3.8 React Native 调试的终端控制台

此时，再次输入搜索内容，例如 `Abc`，在终端控制台中将会打印出详细日志如下：

输入的内容是 `A`

输入的内容是 `Ab`

输入的内容是 `Abc`

实现了完整的搜索功能之后，接下来再美化搜索框的样式。

3.2.3 优化搜索框样式

现在的输入框是直角的长方形，但是为了美观，通常应用输入框的边角都是带有一定弧度的，类似如图 3.9 所示的效果。

为了实现圆角边框的效果，给 `TextInput` 组件的样式添加新的属性 `borderWidth` 和 `borderRadius`，修改 `app.js` 代码如下：

```
01 const styles = StyleSheet.create({
02   // 这里省略了没有修改的代码
03   input: {
04     flex: 1,
05     borderColor: 'gray',
06     borderWidth: 2,
07     borderRadius: 10
08   },
09   // 这里省略了没有修改的代码
10 });
```

重新加载应用后，美化后的搜索框效果如图 3.10 所示。

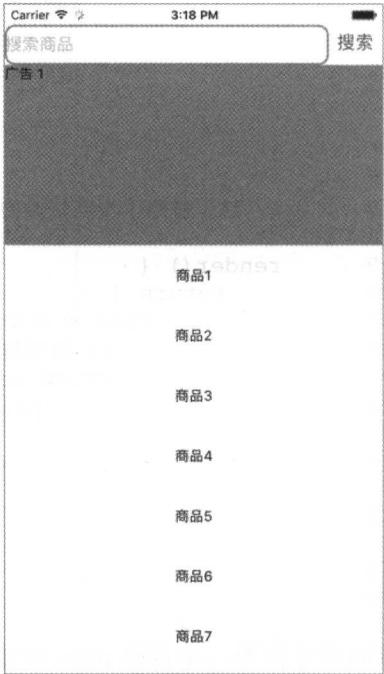


图 3.10 圆角边框的输入框

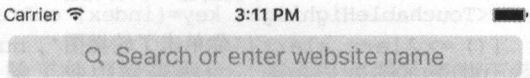


图 3.9 圆角边框的输入框

3.3 完善轮播广告——Image 组件

之前的轮播广告页面显示的是简单的文字和背景色，接下来添加一些好看的图片作为轮播广告。

React Native 中用于图片显示的组件是 `Image`。`Image` 组件可以显示多种不同类型图片，包括网络图片、静态资源、临时的本地图片，以及本地磁盘上的图片（如相册）等。

3.3.1 使用网络图片

这里先使用网络图片来看看 `Image` 的用法和效果。修改 `app.js` 代码如下：

```

01 export default class app extends Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       advertisements: [           // 轮播广告数组
06         { // 数组中的每个成员描述了网络图片的 url
07           url: 'https://img13.360buyimg.com/cms/jfs/t4090/228/1399180862/217278/206073fe/5874e621Nc675c6d0.jpg'
08         }, {
09           url: 'https://img13.360buyimg.com/cms/jfs/t3937/164/1340098884/295670/ca0ebbafe/58703afbN5336c28d.jpg'
10         }, {
11           url: 'https://img14.360buyimg.com/cms/jfs/t3190/189/5382195407/297118/377d637e/586f5b7bN9c81c29c.jpg'
12         }
13       ],
14     };
15   }
16
17   // 这里省略了没有修改的代码
18
19   render() {
20     return (
21       <View style={styles.container}>
22         // 这里省略了没有修改的代码
23         <View style={styles.advertisement}>
24           <ScrollView ref="scrollView"
25             horizontal={true}
26             showsHorizontalScrollIndicator={false}
27             pagingEnabled={true}>
28             {this.state.advertisements.map((advertisement,
29               index) => {
30               return (
31                 <TouchableHighlight key={index} onPress=
32                   (() => Alert.alert('你单击了轮播图', null,
33                     null))>
34                   <Image style={styles.advertisement
35                     Content}
36                     source={{
37                       uri: advertisement.url
38                     }}></Image>

```

```

35         </TouchableHighlight>
36     );
37     }}}
38 </ScrollView>
39 </View>
40 // 这里省略了没有修改的代码
41 </View>
42 );
43 }
44
45 // 这里省略了没有修改的代码
46 }

```

注意：当使用新的模块或组件时（例如这里的 Image）时，首先必须要在文件头导入该模块或组件 `import {Image} from 'react-native'`；否则会发生无法找到变量 Image 的错误。

重新加载应用，此时使用网络图片的轮播广告效果如图 3.11 所示。

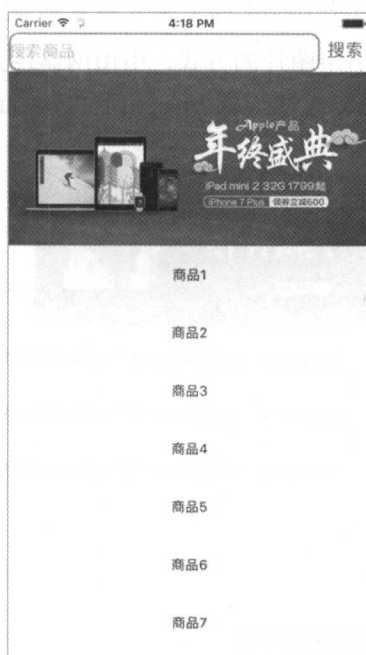


图 3.11 使用网络图片的轮播广告

3.3.2 使用本地图片

除了使用网络图片，还可以将图片资源下载后添加到 ch04 项目中，使用本地图片。

首先，将上面用到的网络图片下载到本地，重命名为 advertisement-image-01.jpg、advertisement-image-02.jpg 以及 advertisement-image-03.jpg，然后复制至 ch04 项目文件夹中，效果如图 3.12 所示。

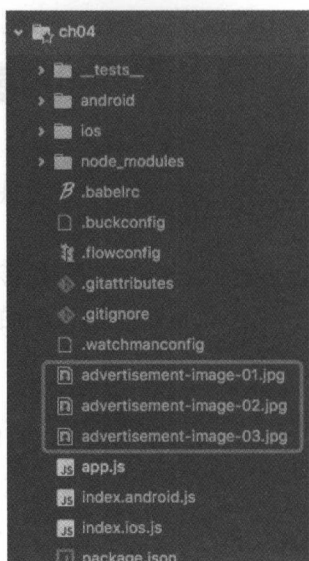


图 3.12 下载并添加图片到 ch04 项目


然后，修改代码中 `Image` 引用图片的方式，引用网络图片是设置 `Image` 组件 `source` 属性的 `url` 值，而引用本地图片可以直接设置 `Image` 组件的 `source` 属性，图片通过 `require` 方式加载，修改后的 `app.js` 代码如下：

```
01 export default class app extends Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       advertisements: [          // 轮播广告数组
06         {
07           image: require('./advertisement-image-01.jpg')
08         }, {
09           image: require('./advertisement-image-02.jpg')
10         }, {
11           image: require('./advertisement-image-03.jpg')
12         }
13       ],
14     };
15   }
16
17   // 这里省略了没有修改的代码
18
19   render() {
20     return (
21       <View style={styles.container}>
22         // 这里省略了没有修改的代码
23         <View style={styles.advertisement}>
24           <ScrollView ref="scrollView"
25             horizontal={true}
26             showsHorizontalScrollIndicator={false}
27             pagingEnabled={true}>
28             {this.state.advertisements.map((advertisement,
29               index) => {
30               return (
31                 <TouchableHighlight key={index} onPress=
32                   {() => Alert.alert('你单击了轮播图', null,
```



```

    null)}}>
31      <Image style={styles.advertisement
      Content}
32        source={advertisement.image}>
33      </Image>
34    </TouchableHighlight>
35  );
36  }}}
37  </ScrollView>
38  </View>
39  // 这里省略了没有修改的代码
40  </View>
41  );
42  }
43
44  // 这里省略了没有修改的代码
45  }
```

 **提示：** 引用本地图片资源时，需要格外注意 `require` 的图片文件路径，否则会发生找不到图片的错误。以上述代码为例，`require('./advertisement-image-01.jpg')`引用的文件路径是指，在 `ch04` 项目根目录下的 `advertisement-image-01.jpg`。

使用本地图片的效果如图 3.13 所示。

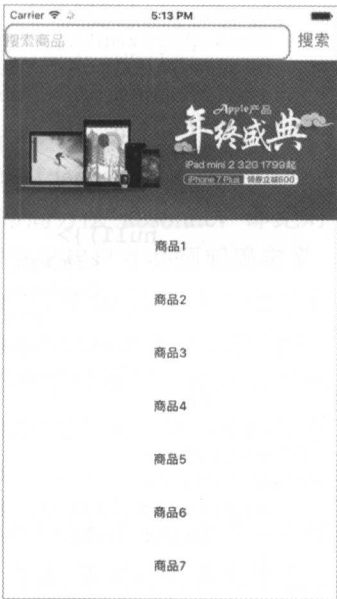


图 3.13 使用本地图片的轮播广告

3.3.3 添加指示器组件

给轮播广告换上了漂亮的图片之后，还差一个效果：当前页面指示器，效果如图 3.14 所示。

从图 3.14 中可以看出，指示器由圆点组成，圆点的个数即页面的数量，并且与当前页面序号相同的圆点会做颜色区分。

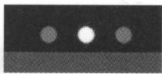


图 3.14 页面指示器

(1) 在了解了指示器的原理之后, 首先定义指示器中圆点的尺寸, 修改 app.js 代码如下:

```
01 const circleSize = 8;
02 const circleMargin = 5;
03
04 export default class app extends Component {
05 // 这里省略了没有修改的代码
```

(2) 在 render() 函数中的轮播广告中添加指示器组件, 代码如下:

```
01 export default class app extends Component {
02 // 这里省略了没有修改的代码
03
04 render() {
05     const advertisementCount = this.state.advertisements.length;
06     // 指示器圆点个数
07     const indicatorWidth = circleSize * advertisementCount +
08     circleMargin * advertisementCount * 2; // 计算指示器的宽度
09     const left = (Dimensions.get('window').width - indicatorWidth)
10     / 2; // 计算指示器最左边的坐标位置
11
12     return (
13         <View style={styles.container}>
14             // 这里省略了没有修改的代码
15             <View style={styles.advertisement}>
16                 <ScrollView ref="scrollView"
17                 horizontal={true}
18                 showsHorizontalScrollIndicator={false}
19                 pagingEnabled={true}>
20                     {this.state.advertisements.map((advertisement,
21                     index) => {
22                         return (
23                             <TouchableHighlight key={index} onPress=
24                             (() => Alert.alert('你单击了轮播图', null,
25                             null))>
26                                 <Image style={styles.advertisement
27                                 Content}
28                                 source={advertisement.image}>
29                                 </Image>
30                                 </TouchableHighlight>
31                             );
32                         )}}
33                     </ScrollView>
34                     <View style={[
35                     styles.indicator, {
36                     left: left
37                     }
38                     ]}>
39                     {this.state.advertisements.map((advertisement,
40                     index) => {
41                         return (<View key={index}
42                         style={(index === this.state.currentPage)
43                         ? styles.circleSelected
44                         : styles.circle}>);
45                     )}}
46                     </View>
47                 </View>
48             // 这里省略了没有修改的代码
49         </View>
50     );
51 }
```

```

44
45      // 这里省略了没有修改的代码
46  }

```

在上述代码中, 首先通过指示器圆点的个数计算出了指示器的宽度, 然后通过屏幕和指示器的宽度, 计算出了指示器最左边的坐标位置。


(3) 修改样式和布局的代码如下:

```

01  const styles = StyleSheet.create({
02      // 这里省略了没有修改的代码
03      indicator: {
04          position: 'absolute',
05          top: 160,
06          flexDirection: 'row'
07      },
08      circle: {
09          width: circleSize,
10          height: circleSize,
11          borderRadius: circleSize / 2,
12          backgroundColor: 'gray',
13          marginHorizontal: circleMargin
14      },
15      circleSelected: {
16          width: circleSize,
17          height: circleSize,
18          borderRadius: circleSize / 2,
19          backgroundColor: 'white',
20          marginHorizontal: circleMargin
21      },
22      // 这里省略了没有修改的代码
23  });
24  });

```

这里我们使用了一种新的布局方法 `absolute`, 即绝对布局。使用绝对布局时, 组件的位置和尺寸必须明确定义, 例如上述代码中 `top`、`left`、`width` 以及 `height`。由于绝对布局并没有 `flex` 布局自适应屏幕的能力, 所以在实际开发中, 使用绝对布局组件的以上属性往往是通过计算动态得到的, 例如上述代码中, 指示器的宽度和最左边的坐标位置都是在获取屏幕宽度后, 计算得到的。

 **提示:** 在实际开发中, 如果使用绝对布局, 千万不要将位置和尺寸定义为固定值, 否则将无法支持不同屏幕的适配。

(4) 添加完指示器并配置好样式后, 运行效果如图 3.15 所示。

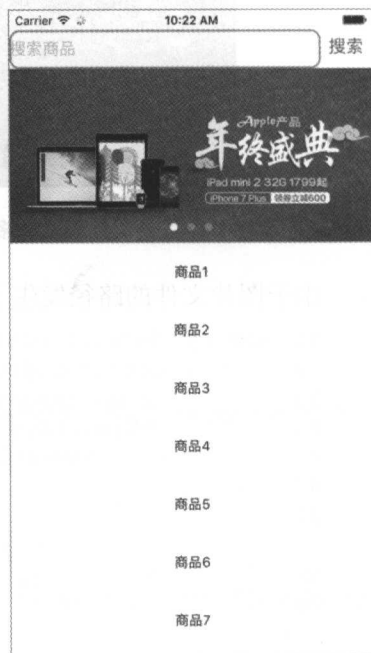


图 3.15 有指示器的轮播广告

3.4 完善商品列表——ListView 组件

在完善了搜索框和轮播广告之后, 对电商应用的首页的改造已经初见成效。最后, 我们要让商品列表的内容也变得更加丰富。

3.4.1 对图片资源进行重构

添加商品图片到 ch04 项目中，相对 ch04 根目录的文件路径为 ./product-image-01.jpg。此时，如果不断地添加图片文件的话，根目录的结构会变得越来越“糟糕”，如图 3.16 所示。

这时需要对图片资源进行一次重构，将所有图片放至专门的 image 文件夹中。重构之后，ch04 项目的文件结构如图 3.17 所示。



图 3.16 根目录文件越来越多的 ch04 项目

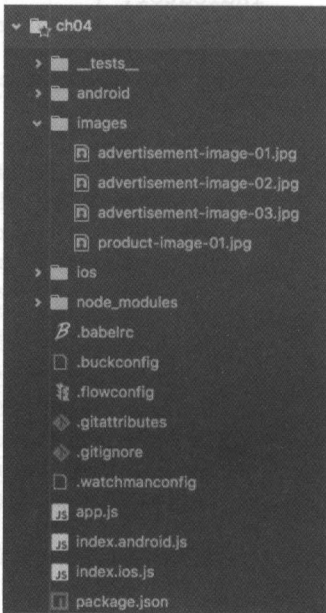


图 3.17 重构图片资源后的 ch04 项目结构

由于图片文件的路径发生了变更，所以还需要修改引用图片的代码如下：

```
01 export default class app extends Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       advertisements: [
06         {
07           image: require('./images/advertisement-image-01.jpg')
08         }, {
09           image: require('./images/advertisement-image-02.jpg')
10         }, {
11           image: require('./images/advertisement-image-03.jpg')
12         }
13       ],
14     };
15   }
16
17   // 这里省略了没有修改的代码
18 }
```

3.4.2 重新定义商品模型

在添加好商品图片之后，第一步就是要修改和重新定义商品模型，修改 `app.js` 代码如下。

提示：在实际开发中，一般都是先定义数据模型，然后再考虑具体功能的实现。

```

01 const ds = new ListView.DataSource({ // 创建 ListView.DataSource 数据源
02   rowHasChanged: (r1, r2) => r1 !== r2
03 });
04
05 export default class app extends Component {
06   constructor(props) {
07     super(props);
08     this.state = {
09       dataSource: ds.cloneWithRows([ // 为数据源传递一个数组
10         {
11           image: require('./images/advertisement-image-01.jpg'),
12           title: '商品 1',
13           subTitle: '描述 1'
14         }, {
15           image: require('./images/advertisement-image-01.jpg'),
16           title: '商品 2',
17           subTitle: '描述 2'
18         }, {
19           // 这里省略了重复的代码
20         }, {
21           image: require('./images/advertisement-image-01.jpg'),
22           title: '商品 10',
23           subTitle: '描述 10'
24         }
25       ],
26     );
27   }
28
29   // 这里省略了没有修改的代码
30 }

```

然后，在 `ListView` 组件的 `_renderRow()` 函数中添加 `Image` 组件，修改 `app.js` 代码如下：

```

01 export default class app extends Component {
02   // 这里省略了没有修改的代码
03
04   _renderRow = (rowData, sectionID, rowID) => {
05     return (
06       <TouchableHighlight onPress={() => Alert.alert('你单击了商品列表', null, null)}>
07         <View style={styles.row}>
08           <Image source={rowData.image}
09             style={styles.productImage}>
10             </Image>
11             <Text style={styles.productTitle}>{rowData.title}</Text>
12             <Text style={styles.productSubTitle}>{rowData.subTitle}</Text>
13           </View>
14         </TouchableHighlight>
15       );

```

```
16     }
17   }
18
19   const styles = StyleSheet.create({
20     // 这里省略了没有修改的代码
21     row: {
22       height: 60,
23       flexDirection: 'row',
24       alignItems: 'center'
25     },
26     productImage: {
27       marginLeft: 10,
28       width: 40,
29       height: 40
30     },
31     productText: {}, // 这里暂时未用到, 且未设置样式
32     productTitle: {}, // 这里暂未设置样式
33     productSubTitle: {} // 这里暂未设置样式
34   });
```

此时, 商品列表的运行效果如图 3.18 所示。



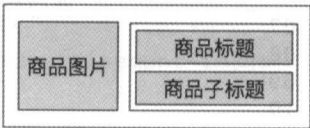
图 3.18 添加了图片和描述的商品列表

3.4.3 商品布局的优化

列表的雏形实现之后, 就可以在此基础上做一些样式和布局上的优化了。优化的目标效果如图 3.19 所示。

按照上述结构, 修改组件代码如下:

```
01 export default class app extends Component {
02   // 这里省略了没有修改的代码
03 }
```



商品列表行结构

图 3.19 商品列表行结构

```

04   _renderRow = (rowData, sectionID, rowID) => {
05     return (
06       <TouchableHighlight onPress={() => Alert.alert('你单击了商品列表', null, null)}>
07         <View style={styles.row}>
08           <Image source={rowData.image}
09             style={styles.productImage}>
10             </Image>
11           <View style={styles.productText}> // flexDirection
            默认为"column"
12             <Text style={styles.productTitle}>{rowData.title}</Text>
13             <Text style={styles.productSubTitle}>
14               {rowData.subTitle}</Text>
15             </Text>
16           </View>
17         </View>
18       </TouchableHighlight>
19     );
20   }
21
22   // 这里省略了没有修改的代码
23 }


```

此时, 应用的运行效果如图 3.20 所示。



图 3.20 优化后的商品列表

完成了列表的基本布局之后, 接着优化样式。修改 app.js 代码如下。

 **提示:** 在实际开发中, 先搭好架子再优化细节是一个良好且高效的开发习惯。

```

01   const styles = StyleSheet.create({
02     // 这里省略了没有修改的代码
03     row: {
04       height: 60,

```



```

05         flexDirection: 'row',
06         backgroundColor: 'white'
07     },
08     productImage: {
09         width: 40,
10         height: 40,
11         marginLeft: 10,
12         marginRight: 10,
13         alignSelf: 'center'
14     },
15     productText: {
16         flex: 1,
17         marginTop: 10,
18         marginBottom: 10
19     },
20     productTitle: {
21         flex: 3,
22         fontSize: 16
23     },
24     productSubTitle: {
25         flex: 2,
26         fontSize: 14,
27         color: 'gray'
28     }
29 });

```

优化列表样式和布局后的效果如图 3.21 所示。



图 3.21 优化样式和布局后的商品列表

最后，为列表添加分割线。幸运的是，`ListView` 组件已经为开发者提供了方法，即 `renderSeparator()` 函数，所以只要实现该函数即可。修改 `app.js` 代码如下：

```

01 export default class app extends Component {
02     // 这里省略了没有修改的代码
03
04     render() {
05         // 这里省略了没有修改的代码

```

```

06
06     return (
07         <View style={styles.container}>
08             // 这里省略了没有修改的代码
09             <View style={styles.products}>
10                 <ListView dataSource={this.state.dataSource}
11                     renderRow={this._renderRow}
12                     renderSeparator={this._renderSeparator}/>
13                     // 渲染分割线
14             </View>
15         </View>
16     );
17
18 // 这里省略了没有修改的代码
19
20 _renderSeparator(sectionID, rowID, adjacentRowHighlighted) {
21     return (
22         <View key={`_${sectionID}-${rowID}`} style={styles.divider}>
23             </View>
24     );
25 }
26
27 const styles = StyleSheet.create({
28     // 这里省略了没有修改的代码
29     divider: {
30         height: 1,
31         width: Dimensions.get('window').width - 5,
32         marginLeft: 5,
33         backgroundColor: 'lightgray'
34     },
35     // 这里省略了没有修改的代码
36 }

```

重新加载应用，添加分割线的商品列表效果如图 3.22 所示。




图 3.22 添加分割线的商品列表

至此，电商 App 的首页已经焕然一新。

3.5 拖曳刷新列表——RefreshControl 组件

在 3.4 节中，我们完善了商品列表的功能：不仅优化了列表的布局，还添加了分割线等效果。不过，该 App 还缺少一个常用的功能，那就是拖曳刷新。虽然也可以添加一个“刷新”按钮用于响应用户请求，但是用户体验却没有拖曳好，而且，由于现在移动开发设备屏幕通常较小，额外添加按钮对界面设计影响也较大。

 **小知识：**用户体验（User Experience，简称 UX 或 UE）是涉及一个人使用一个特定产品或系统或服务的有关行为、态度与情绪。用户体验包括实际、体验、情感、有意义、有价值的人机交流，以及产品所有权方面的问题。此外还包括系统方面，例如实用、易用性和效率。

为了实现拖曳刷新列表的效果，需要使用一个新的 React Native 组件：RefreshControl。RefreshControl 组件用在 ScrollView 或 ListView 内部，为其添加下拉刷新的功能。

在使用 RefreshControl 组件之前，首先要在 this.state 中添加一个是否正在刷新中的标志。

（1）修改 app.js 代码如下：

```
01 export default class app extends Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       isRefreshing: false           // 是否正在刷新中的标志
06     };
07   }
08
09   // 这里省略了没有修改的代码
10 }
```

（2）在 ListView 组件中添加 RefreshControl。修改 ListView 相关代码如下：

```
01 export default class app extends Component {
02   // 这里省略了没有修改的代码
03
04   render() {
05     // 这里省略了没有修改的代码
06
07     return (
08       <View style={styles.container}>
09         // 这里省略了没有修改的代码
10         <View style={styles.products}>
11           <ListView dataSource={this.state.dataSource}
12             renderRow={this._renderRow}
13             renderSeparator={this._renderSeparator}
14             refreshControl={this._renderRefreshControl()} />
15         </View>
16       </View>
17     );
18   }
```

```

18
19 // 这里省略了没有修改的代码
20 _renderRefreshControl() {
21   return (
22     <RefreshControl
23       refreshing={this.state.isRefreshing} // 通过 this.state.
        isRefreshing 设置 RefreshControl 是否正在刷新
24       tintColors={ ['#FF0000']}
25       title={'正在刷新数据, 请稍后...'}
26       titleColor={'#0000FF'}>
27     </RefreshControl>
28   );
29 }
30 }

```

重新记载应用后, 下拉拖曳列表, 可以看到添加的 RefreshControl 效果如图 3.23 所示。



图 3.23 添加 RefreshControl 的商品列表

(3) 接着, 需要为 RefreshControl 添加状态变化的逻辑, 修改 app.js 代码如下:

```

01 export default class app extends Component {
02   // 这里省略了没有修改的代码
03
04   _renderRefreshControl() {
05     return (
06       <RefreshControl
07         refreshing={this.state.isRefreshing}
08         onRefresh={this._onRefresh} // 刷新时调用的 onRefresh() 方法
09         tintColors={ ['#FF0000']}
10         title={'正在刷新数据, 请稍后...'}
11         titleColor={'#0000FF'}>
12       </RefreshControl>
13     );
14   }
15 }

```

```

16   _onRefresh = () => {
17     this.setState({isRefreshing: true});           // 设置状态为正在刷新
18
19     setTimeout(() => {
20       this.setState({isRefreshing: false});        // 设置状态为结束刷新
21     }, 2000);                                       // 定时器时间间隔 2 秒
22   }
23 }

```

这时重新下拉拖曳列表然后松开，列表就处于刷新状态了。等待时间间隔 2 秒钟后，列表又会恢复初始状态。

另外，需要提醒读者的是，与 `StatusBar` 组件类似，`RefreshControl` 在不同平台下的效果也是不完全相同的，例如，上述实现在 Android 设备上的运行效果如图 3.24 所示。



图 3.24 `RefreshControl` 在 Android 上的效果

(4) 在完成拖曳刷新的响应之后，还可以进一步完善：不仅仅是等待 2 秒钟，还可以更新商品数据并刷新列表。修改 `app.js` 代码如下：

```

01 export default class app extends Component {
02   // 这里省略了没有修改的代码
03
04   _onRefresh = () => {
05     this.setState({isRefreshing: true});
06
07     setTimeout(() => {
08       const products = Array.from(new Array(10)).map((value,
09         index) => ({
10         image: require('./images/product-image-01.jpg'),
11         title: '新商品' + index,
12         subTitle: '新商品描述' + index
13       }));
14       this.setState({isRefreshing: false, dataSource: ds.clone
15         WithRows(products)});

```

```
14      }, 2000);
15    }
16  }
```

再次下拉拖曳刷新列表，等待时间间隔 2 秒后，商品列表也更新了，如图 3.25 所示。



图 3.25 拖曳刷新列表

3.6 添加页面跳转功能——Navigator 组件

React Native 实现页面跳转的组件有 Navigator 以及 NavigatorIOS，和前面介绍过的 ViewPagerAndroid 问题一样，为了考虑平台兼容性和代码复用性，这里使用同时支持 iOS 和 Android 的 Navigator 组件。

在正式使用 Navigator 之前，首先需要对现有的 ch04 项目做一些简单的重构：将首页的实现移植到新建的 home.js 文件中，以便后面的逻辑改动。

(1) 新建 home.js 文件，将 app.js 的内容全部复制至 home.js 文件中。同时，不要忘记修改 home.js 文件中组件的名称。修改 home.js 代码如下：

```
01 export default class home extends Component { // 将组件名称从"app"修改成"home"
```

(2) 修改 app.js 文件的代码如下：

```
01 import React, {Component} from 'react';
02 import {View, Navigator} from 'react-native';
03
04 import Home from './home';
05
06 export default class app extends React.Component {
07   render() {
08     return (
```

```

09         <Navigator
10             initialRoute=://{
11                 name: 'home',
12                 component: Home
13             }
14             configureScene={(route) => {
15                 return Navigator.SceneConfigs.FloatFromRight;
16             }}
17             renderScene={(route, navigator) => {
18                 const Component = route.component;
19                 return <Component {...route.params} navigator={navigator}/>
20             }
21         }
22     }
23 }

```

重新加载应用，保证运行效果和重构前完全一致，如图 3.26 所示。



图 3.26 代码重构后的运行效果

对于上述代码，读者可能会有很多疑惑，这里来一一说明。

Navigator 组件的 `initialRoute` 属性定义了应用启动时加载的路由（route），而路由是 Navigator 组件用来识别渲染场景的一个对象，简单来说，`initialRoute` 中定义的组件就是应用第一个要显示的页面，这就是首页 `home.js`。

Navigator 组件的 `configureScene` 属性定义了页面之间跳转的动画，除了代码中的 `Navigator.SceneConfigs.FloatFromRight` 外，还有很多 React Native 已经为我们定义好的动画，包括：

- `FloatFromRight`;
- `FloatFromLeft`;
- `FloatFromBottom`;
- `FloatFromBottomAndroid`;

- FadeAndroid;
- HorizontalSwipeJump;
- HorizontalSwipeJumpFromRight;
- VerticalUpSwipeJump;
- VerticalDownSwipeJump。

提示：动画效果也考虑到了平台差异性，这点可以从动画命名可以看出，例如只支持 Android 的动画有 FloatFromBottomAndroid 以及 FadeAndroid 等。

注意：由于动画效果暂时无法在书中呈现给读者，所以读者感兴趣的话，可以自己动手运行应用体验不同的动画效果。

Navigator 组件的 renderScene()函数应该是最容易令人困惑的地方了，为此，可以使用 React Native 调试来一看究竟。在该函数中添加断点并重新运行应用，效果如图 3.27 所示。

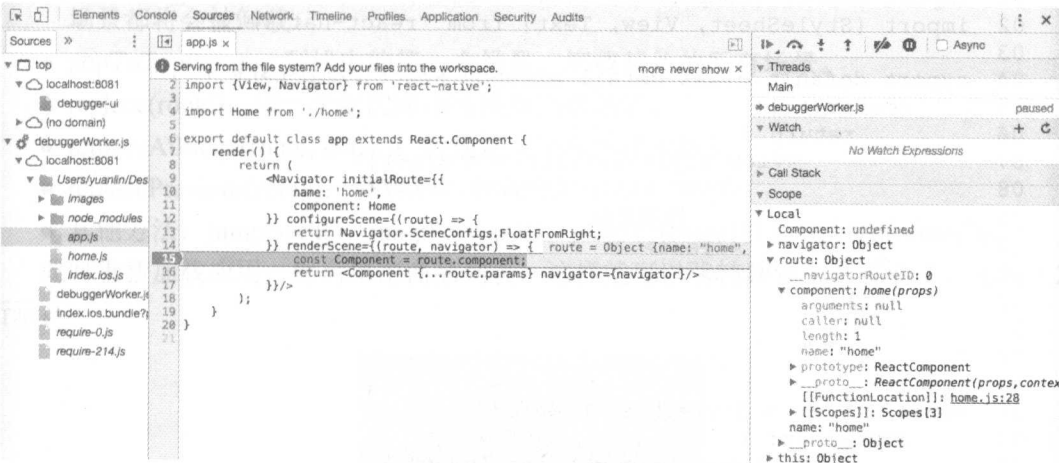


图 3.27 使用调试断点来查看 renderScene 运行机制

这里需要重点关注的是右侧调试区域的变量值，如图 3.28 所示。

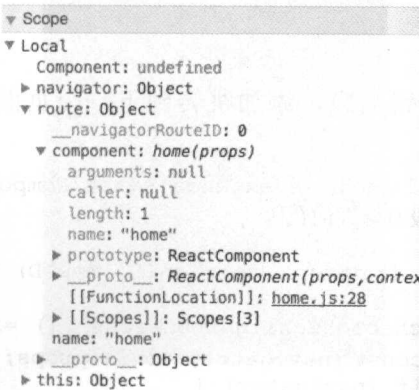


图 3.28 renderScene 参数的值

从图 3.28 中的 name 可以看出，此时的 route 里的 name 和 component 就是在 initialRoute

属性中传递的 `home` 和 `home` 组件，所以 `app.js` 文件中这行代码：

```
01 return <Component {...route.params} navigator={navigator}/>
```

的作用就是返回 `home` 组件。当然，这里不仅返回了 `home` 组件，还将路由（`route`）的参数以及 `navigator` 参数也传递给了 `home` 组件：通过 `{...route.params}` 以及 `navigator= {navigator}`，于是，在 `home` 组件中就可以使用 `this.props.navigator` 来获取 `navigator`。

3.7 二级页面的跳转——TouchableOpacity 组件

理解了 `Navigator` 的基本用法之后，下一步，添加一个新的组件，以便实现二级页面跳转的效果。

（1）添加新的文件 `detail.js`，并在该文件中创建 `detail` 组件，代码如下：

```
01 import React, {Component} from 'react';
02 import {StyleSheet, View, Text} from 'react-native';
03
04 export default class detail extends React.Component {
05   render() {
06     return (
07       <View style={styles.container}>
08         <Text style={styles.text}>
09           详情页面
10         </Text>
11       </View>
12     );
13   }
14 }
15
16 const styles = StyleSheet.create({
17   container: {
18     flex: 1, backgroundColor: 'gray',
19     justifyContent: 'center',
20     alignItems: 'center'
21   },
22   text: {
23     fontSize: 20
24   }
25 });
```

（2）在首页中修改按钮响应，添加跳转到下一个页面的入口。修改 `home.js` 中 `_renderRow` 的代码如下：

```
01 export default class home extends React.Component {
02   // 这里省略了没有修改的代码
03
04   _renderRow = (rowData, sectionID, rowID) => {
05     return (
06       <TouchableHighlight onPress={() => {
07         const {navigator} = this.props; // 从 props 获取 navigator
08         if (navigator) {
09           navigator.push({name: 'detail', component: Detail});
10         }
11       }}>
12       <View style={styles.row}>
13         <Image source={rowData.image}
```

```

14         style={styles.productImage}></Image>
15         <View style={styles.productText}>
16           <Text style={styles.productTitle}>{rowData.title}
17           </Text>
18           <Text style={styles.productSubTitle}>{rowData.
19             subTitle}</Text>
20         </View>
21       </View>
22     </TouchableHighlight>
23   );
24 }

```

在使用 `this.props.navigator` 获取 `Navigator` 之后,可以使用 `Navigator` 的如下接口来进行场景的切换。

- `push(route)`: 跳转到新的场景, 并且将场景入栈。
- `pop`: 跳转到上一个场景, 并且将当前场景出栈。
- `popToRoute(route)`: `pop` 到路由指定的场景, 在整个路由栈中, 处于指定场景之后的场景都将会被卸载。
- `popToTop()`: `pop` 到栈中的第一个场景, 卸载其他的所有场景。
- `replace(route)`: 用一个新的路由替换掉当前场景。
- `replaceAtIndex(route, index)`: 替换指定序号的路由场景。
- `replacePrevious(route)`: 替换上一个场景。
- 其他方法: `jumpBack()`、`jumpForward()`、`jumpTo(route)`以及 `resetTo(route)`等。

此时重新加载应用, 然后单击商品列表, 这样就可以跳转到下一个页面了, 如图 3.29 所示。

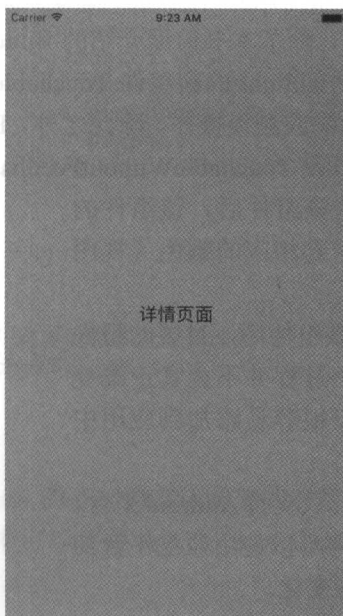


图 3.29 二级详情页面

(3) 那么如何返回到上一个页面呢? 答案就是使用 `Navigator` 的 `pop()`方法。修改 `detail.js` 代码如下:

```

01 import React, {Component} from 'react';
02 import {StyleSheet, View, Text, TouchableOpacity} from 'react-native';
03
04 export default class detail extends React.Component {
05   render() {
06     return (
07       <View style={styles.container}>
08         <TouchableOpacity onPress={this._pressBackButton.bind
09           (this)}>
10           <Text style={styles.back}>返回</Text>
11         </TouchableOpacity>
12         <Text style={styles.text}>
13           详情页面
14         </Text>
15       </View>
16     );
17   }
18   _pressBackButton() {
19     const {navigator} = this.props;
20     if (navigator) {
21       navigator.pop();
22     }
23   }
24 }
25
26 const styles = StyleSheet.create({
27   // 这里省略了没有修改的代码
28   back: {
29     fontSize: 20,
30     color: 'blue'
31   }
32 });

```


为了实现“返回”按钮功能，除了本书前面介绍的 Button 和 TouchableHighlight 组件，这里使用了一个类似 TouchableHighlight 的新组件 TouchableOpacity。TouchableOpacity 同样用于封装组件，使其可以正确响应触摸操作。除此之外，React Native 的 Touchable* 组件还有 TouchableNativeFeedback 以及 TouchableWithoutFeedback，它们到底有什么区别呢？

- **TouchableHighlight**: 单击该组件后，该组件的不透明度会降低同时会看到相应的颜色（视图变暗或者变亮）。
- **TouchableOpacity**: 单击该组件后，封装的组件的不透明度会降低。这个过程并不会真正改变组件层级，大部分情况下很容易添加到应用中而不会带来副作用。
- **TouchableNativeFeedback**: 只支持 Android 平台，在 Android 平台上该组件可以使用原生平台的状态资源来显示触摸状态变化。
- **TouchableWithoutFeedback**: 单击该组件后，该组件没有任何反应和变化，所以不推荐使用。

此时，重新加载应用，然后单击图 3.30 中的“返回”按钮可以返回到首页了。



图 3.30 单击“返回”按钮返回到首页

提示：更多页面间的跳转和动画内容限于篇幅，这里不再一一详细介绍，感兴趣的读者可以自己修改代码体验效果。

3.8 实现页面间的数据传递

跳转和返回的效果实现了，那么，如何实现页面间的数据传递和通信呢？

其实，从上述代码中读者已经可以发现：React Native 使用 props 来实现页面间数据传递和通信。在 React Native 中，有两种方式可以存储和传递数据，即 props（属性）以及 state（状态），其中：

- props 通常是在父组件中指定的，而且一经指定，在被指定的组件的生命周期中则不再改变。
- state 通常是用于存储需要改变的数据，并且当 state 数据发生更新时，React Native 会刷新界面。

了解了 props 与 state 的区别之后，读者应该知道，要将首页的数据传递到下一个页面，需要使用 props。所以修改 home.js 代码如下：

```
01 export default class home extends React.Component {
02   // 这里省略了没有修改的代码
03
04   _renderRow = (rowData, sectionID, rowID) => {
05     return (
06       <TouchableHighlight onPress={() => {
07         const {navigator} = this.props; // 从 props 获取 navigator
08         if (navigator) {
09           navigator.push({
10             name: 'detail',
11             component: Detail,
12             params: {
13               productTitle: rowData.title
14             } // 通过 params 传递 props
15           });
16         }
17       }) >>
18       // 这里省略了没有修改的代码
19     </TouchableHighlight>
20   );
21 }
22 }
```

在 home.js 中，为 Navigator 的 push() 方法添加的参数 params，会当做 props 传递到下一个页面，因此，在 detail.js 中可以使用 this.props.productTitle 来获得首页传递的数据。修改 detail.js 代码如下：

```
01 export default class detail extends React.Component {
02   render() {
03     return (
04       <View style={styles.container}>
05         <TouchableOpacity onPress={this._pressBackButton.bind
06           (this)}>
```

```

06         <Text style={styles.back}>返回</Text>
07     </TouchableOpacity>
08     <Text style={styles.text}>
09         {this.props.productTitle}
10     </Text>
11 </View>
12 );
13 }
14
15 // 这里省略了没有修改的代码
16 }

```

重新加载应用，当再次单击商品列表时，详情页面将显示单击的商品名称，效果如图 3.31 所示。

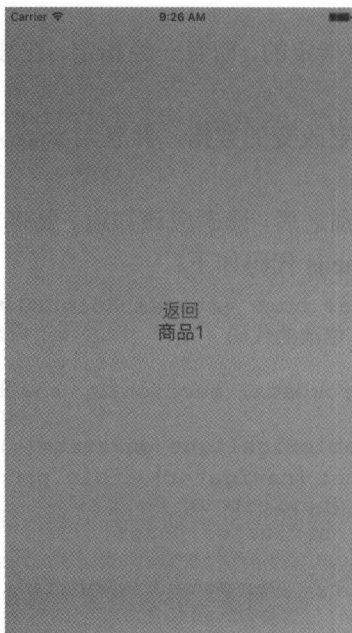


图 3.31 详情页面显示单击的商品名称

这样，一个完整的页面跳转和页面间数据传递的功能就实现了。

3.9 小 结

截止本章，我们不仅开发了一个电商应用，还对已有的应用进行了以下优化

- 代码重构：包括组件的复用、逻辑的简化以及扩展性的优化。
- 样式美化：自定制了组件的样式。
- 功能完善：为轮播广告添加指示器效果，为商品列表添加图片和详细说明，为应用添加更多 React Native 提供的组件。

本书通过电商 App 的改造，学习了多个组件的使用，并带入了实际项目开发中的一些技巧，相信对初学者的学习有更好的指导作用。

第 4 章 React Native 的组件（2）

应用的功能愈加完善，所使用到的 React Native 组件也越来越多，截止到目前，我们使用的组件已经有：

- View;
- Text;
- TextInput;
- Button;
- ScrollView;
- ListView;
- Alert;
- TouchableHighlight;
- StatusBar;
- RefreshControl;
- Image;
- Navigator;
- TouchableOpacity。

不过，这些只是 React Native 庞大组件库的“冰山一角”，因此，本章来学习更多的组件使用情况。

本章主要内容有：

- 介绍更多 React Native 自带的组件。
- 学会使用第三方组件。

4.1 只支持特定平台的组件

迄今，我们使用的都是通用组件，即同时支持 iOS 和 Android 平台的组件，而还有一类组件是我们避而不谈的，那就是特定组件，它们只支持特定的平台。本节就来讲解一些特定组件。

4.1.1 实现多页面分页 TabBarIOS/ViewPagerAndroid

只支持特定平台的组件，例如以下两种。

- 只支持 iOS 的组件：DatePickerIOS、PickerIOS、ProgressViewIOS、SegmentedControlIOS 以及 TabBarIOS 等。

- 只支持 Android 的组件: DrawerLayoutAndroid、ProgressBarAndroid、ToolBarAndroid 以及 ViewPagerAndroid 等。

这里就以 `TabBarIOS` 和 `ViewPagerAndroid` 为例, 虽然它们是支持不同平台的 `React Native` 组件, 但是实现的功能是类似的, 即实现多页面分页的效果。

由于不同平台使用的组件不同, 这里首先要对 `ch04` 项目的代码做一些调整。

(1) 新建 `main.ios.js` 和 `main.android.js` 两个文件, `ch04` 项目结构如图 4.1 所示。



图 4.1 重构后的 `ch04` 项目结构

(2) 给这两个文件添加如下代码:

```
01 import React, {Component} from 'react';
02
03 import Home from './home';
04
05 export default class main extends React.Component {
06   render() {
07     return (
08       <Home navigator={this.props.navigator}></Home>
09     );
10   }
11 }
```

(3) 将 `app.js` 文件中的 `home` 组件替换成 `main` 组件。修改 `app.js` 代码如下:

```
01 import React, {Component} from 'react';
02 import {View, Navigator} from 'react-native';
03
04 import Main from './main';
05
06 export default class app extends React.Component {
07   render() {
08     return (
09       <Navigator
10         initialRoute={{
10           name: 'main',
11           component: Main
12         }}
13       >
14         <ConfigureScene={ (route) => {
14           return Navigator.SceneConfigs.FloatFromRight;
15         }}
16       >
17         <RenderScene={ (route, navigator) => {
17           const Component = route.component;
17           return <Component {...route.params} navigator=
17             {navigator}/>
18         }}/>
19     );
20   }
21 }
22 }
```

(4) 调整过后, 重新运行应用, 保证 `iOS App` 和 `Android App` 运行仍然正常。

读者可能会好奇, `main` 组件的文件名不是 `main.ios.js` 和 `main.android.js` 吗? `React Native` 项目怎么还能正确寻找到 `main` 组件呢? 下面就来介绍下 `React Native` 是如何实现平台特定代码的。

对于 `main.ios.js` 和 `main.android.js` 文件名中带有 `.ios` 和 `.android` 这样平台扩展名的文件,

React Native 会自动检测某个文件是否具有 .ios. 或是 .android. 的扩展名, 然后根据当前运行的平台加载正确对应的文件。

例如, ch04 项目中有如下两个文件:

```
main.ios.js
main.android.js
```

这样命名组件后就可以在其他组件中直接引用, 而无须关心当前运行的平台是哪个平台。React Native 会根据运行平台的不同引入正确对应的组件。

```
01 import Main from './main';
```

- 在 iOS 平台上: 上述代码引用的是 main.ios.js 文件。
- 在 Android 平台上: 上述代码引用的是 main.android.js 文件

(5) 添加一个新的“更多”页面, 新建 more.js 文件并添加代码如下:

```
01 import React, {Component} from 'react';
02 import {StyleSheet, View, Text} from 'react-native';
03
04 export default class more extends React.Component {
05   render() {
06     return (
07       <View style={styles.container}>
08         <Text style={styles.text}>
09           更多页面
10         </Text>
11       </View>
12     );
13   }
14 }
15
16 const styles = StyleSheet.create({
17   container: {
18     flex: 1,
19     justifyContent: 'center',
20     alignItems: 'center'
21   },
22   text: {
23     fontSize: 20
24   }
25 });
```

(6) 有了“更多”页面, 就很容易添加 TabBarIOS 组件了, 修改 main.ios.js 代码如下:

```
01 import React, {Component} from 'react';
02 import {TabBarIOS} from 'react-native';
03
04 import Home from './home';
05 import More from './more';
06
07 export default class main extends React.Component {
08   constructor(props) {
09     super(props);
10     this.state = {
11       selectedTab: 'home'
12     }
13   }
14
15   render() {
16     return (
```

```

17      <TabBarIOS unselectedTintColor="gray" // 未选中标签的颜色
18                tintTintColor="white"      // 选中的标签颜色
19                barTintColor="orange">    // 标签栏的背景色
20      <TabBarIOS.Item title="首页"
21                    icon={require('./images/icon-home.png')} // 图标
22                    selected={this.state.selectedTab === 'home'}
23                    onPress={() => {
24                      this.setState({selectedTab: 'home'});
25                    }}>
26        <Home navigator={this.props.navigator}></Home>
27      </TabBarIOS.Item>
28      <TabBarIOS.Item systemIcon="more" // 使用 React Native
                                         系统图标
29                    badge={2}          // 提醒的数量
30                    selected={this.state.selectedTab === 'more'}
31                    onPress={() => {
32                      this.setState({selectedTab: 'more'});
33                    }}>
34        <More navigator={this.props.navigator}></More>
35      </TabBarIOS.Item>
36    </TabBarIOS>
37  );
38  }
39  }

```

(7) 在运行上述代码之前，不要忘记在 ch04 项目中添加 TabBarIOS.Item 的 icon 所使用的图片资源 ./images/icon-home.png。

重新加载应用，可以看到 TabBarIOS 组件的效果如图 4.2 所示。



图 4.2 TabBarIOS 组件

(8) 成功添加 TabBarIOS 组件后，再来看看 viewPagerAndroid 组件的效果。修改 main.android.js 代码如下：

```

01 import React, {Component} from 'react';
02 import {StyleSheet, View, Text, ViewPagerAndroid} from 'react-native';
03
04 import Home from './home';
05 import More from './more';
06
07 export default class main extends React.Component {
08   render() {
09     return (
10       <ViewPagerAndroid style={styles.viewPager} initialPage={0}>
11         <View style={styles.pageStyle}>
12           <Home navigator={this.props.navigator}></Home>
13         </View>
14         <View style={styles.pageStyle}>
15           <More navigator={this.props.navigator}></More>
16         </View>
17       </ViewPagerAndroid>
18     );
19   }
20 }
21
22 const styles = StyleSheet.create({
23   viewPager: {
24     flex: 1
25   }
26 });

```

此时, ViewPagerAndroid 的效果如图 4.3 所示。

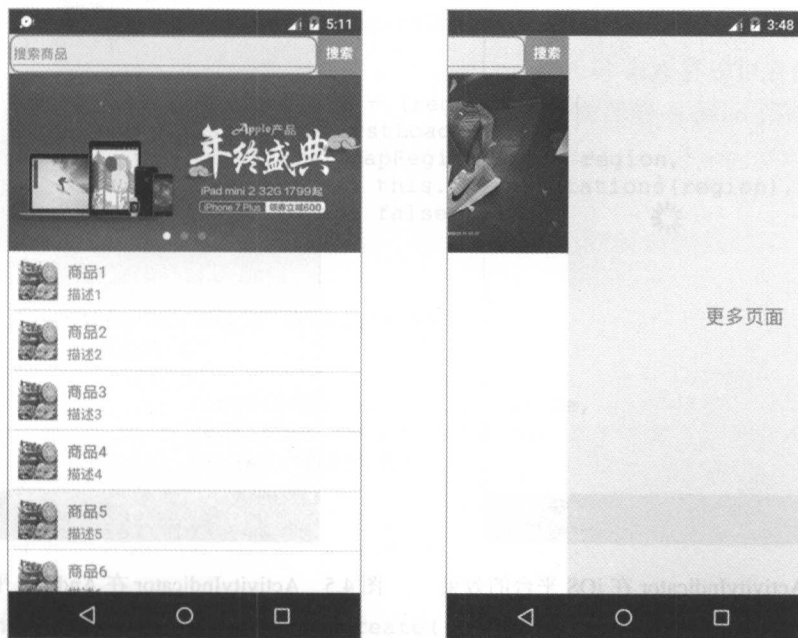


图 4.3 ViewPagerAndroid 组件

通过上述 TabBarIOS/ViewPagerAndroid 的例子,可能读者会有这样的感受: React

Native 开发首先必须要考虑平台复用性，但是在一些情况下，针对不同平台仍然会有差异性的实现，在保证设计和逻辑尽可能复用的前提下，React Native 对于跨平台的开发成本相对于原生开发来说还是比较小的。

4.1.2 加载指示器——ActivityIndicator

ActivityIndicator 组件是一个加载指示器，俗称“转菊花”。其实 RefreshControl 组件中就包含了 ActivityIndicator（详情可参考 3.5 节），这里单独添加 ActivityIndicator 看一下效果。修改 more.js 代码如下：

```
01 export default class more extends React.Component {
02   render() {
03     return (
04       <View style={styles.container}>
05         <ActivityIndicator color="purple" size="large"/>
06       </View>
07     );
08   }
09 }
```

重新加载应用，iOS App 运行效果如图 4.4 所示。

这里需要读者注意的是，ActivityIndicator 组件在不同平台的表现是不尽相同的，例如 Android App 的效果如图 4.5 所示。

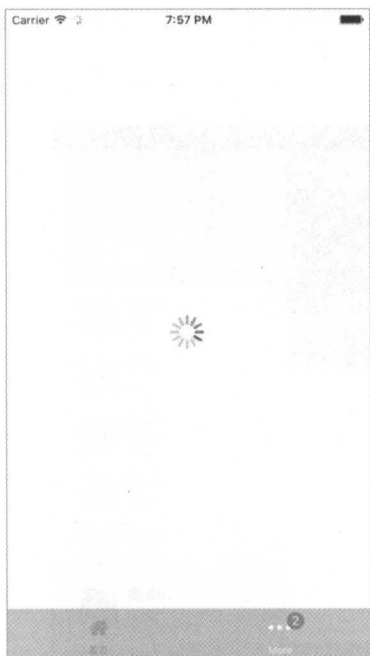


图 4.4 ActivityIndicator 在 iOS 平台的效果

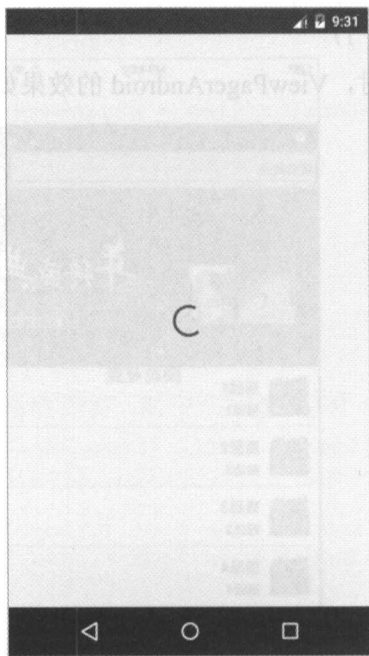



图 4.5 ActivityIndicator 在 Android 平台的效果

 **提示：**上述例子展示的都是组件基本属性的用法，本书不可能将官方文档或源代码中所有的属性全部介绍完，所以在了解基本属性用法的前提下，读者想要实现更复杂的效果，可以参考官方文档或源代码。

4.1.3 地图——MapView

地图是应用中一个很常用的功能，React Native 为开发者提供了 MapView 组件来实现地图功能。现在用下面的例子来展示 MapView 的使用。修改 more.js 代码如下：

```

01 export default class more extends React.Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       isFirstLoad: true,
06       mapRegion: undefined,
07       mapRegionInput: undefined,
08       annotations: []
09     }
10   }
11
12   render() {
13     return (
14       <View style={styles.container}>
15         <MapView style={styles.map}
16           onRegionChange={this._onRegionChange}
17                               // 位置更新时回调
18           onRegionChangeComplete={this._onRegionChangeComplete}
19           region={this.state.mapRegion} // 设置 MapView 位置
20           annotations={this.state.annotations}/>
21       </View>
22     );
23
24     _onRegionChange = (region) => {
25       this.setState({mapRegionInput: region});
26     }
27
28     _onRegionChangeComplete = (region) => {
29       if (this.state.isFirstLoad) {
30         this.setState({mapRegionInput: region,
31           annotations: this._getAnnotations(region),
32           isFirstLoad: false});
33       }
34     }
35
36     _getAnnotations = (region) => {
37       return [
38         {
39           longitude: region.longitude,
40           latitude: region.latitude,
41           title: '你的位置'
42         }
43       ];
44     }
45   }
46
47   const styles = StyleSheet.create({
48     // 这里省略了没有修改的代码
49     map: {
50       width: Dimensions.get('window').width,
51       height: Dimensions.get('window').height

```

```


52     }
53   }

```

上述代码中 MapView 组件各个属性的作用如下。

- **region** 属性用来设置显示的地图区域。
- **annotations** 属性用来设置大头针的位置。
- **onRegionChange** 是在位置发生改变时的回调，回调中包含了最新的位置信息，用于更新地图。


此时重新加载应用，可以看到 MapView 的效果如图 4.6 所示。

 **注意：**使用地图服务的前提是，打开设备的定位服务，并且允许引用使用设备的定位服务。

但是这里却出现一个警告，查看警告详情可以发现，原来官方会在 React Native 0.42 版本中彻底废弃 MapView，所以，提示开发者使用第三方的替代方案 react-native-maps (<https://github.com/airbnb/react-native-maps>)。



图 4.6 MapView 组件

 **提示：**网络上有一些优秀的第三方组件或替代方案，可以达到更好的效果或更高的开发效率，甚至代替 React Native 组件（例如上面的 react-native-maps 就取代了 MapView 组件）。

4.1.4 渲染——Picker

Picker 组件可以在 iOS 和 Android 平台上渲染原生的选择器 (Picker)。下面还是来做一个演示，修改 more.js 代码如下：

```

01 export default class more extends React.Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       language: 'java'
06     }
07   }
08
09   render() {
10     return (
11       <View style={styles.container}>
12         <Picker
13           style={styles.picker}
14           selectedValue={this.state.language}
15           onChange={ (lang) => this.setState({language: lang}) }>
16           <Picker.Item label="Java" value="java"/>
17           <Picker.Item label="JavaScript" value="javascript"/>
18         </Picker>
19       </View>
20     );
21   }
22 }

```



```

23
24 const styles = StyleSheet.create({
25   // 这里省略了没有修改的代码
26   picker: {
27     width: 200,
28     height: 200
29   }
30 }

```

Picker 组件在 iOS 和 Android 平台上的效果如图 4.7 所示。

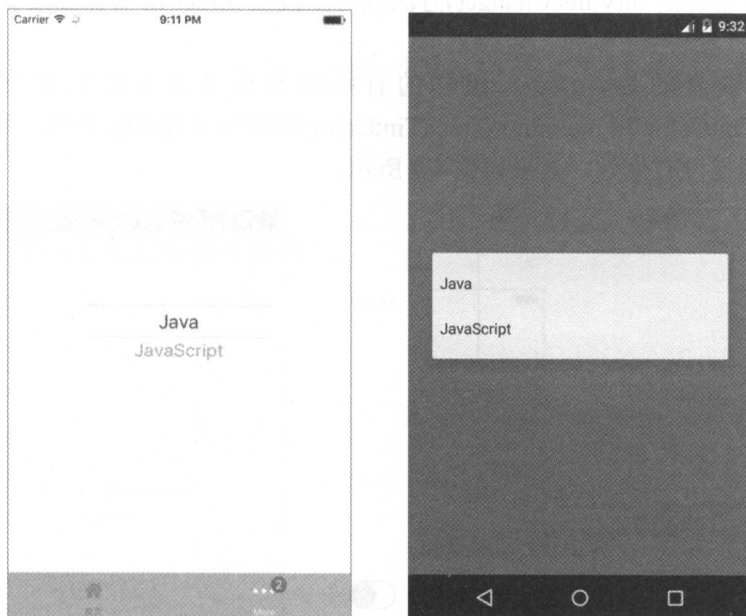


图 4.7 iOS 和 Android 平台上的 Picker 效果

4.1.5 选择范围——Slider

Slider 是一个用于选择范围的组件，用法比较简单。修改 more.js 代码如下：

```

01 export default class more extends React.Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       sliderValue: 5
06     }
07   }
08
09   render() {
10     return (
11       <View style={styles.container}>
12         <Slider minimumValue={0} // 最小值
13           style={{ width: 200 }}
14           step={1} // 步长, 在 minimumValue 和 maximumValue 之间
15           maximumTrackTintColor='red' // Slider 滑道右侧的颜色
16           minimumTrackTintColor='blue' // Slider 滑道左侧的颜色
17           maximumValue={10} // 最大值

```

```

18             value={this.state.sliderValue }
19             // Slider 滑块的初始位置
20             onChange={ (value) => this.setState({sliderValue:
21               value}) }/>
22             <Text>Slider 值: {this.state.sliderValue}</Text>
23           </View>
24         );
25       }
26     }

```

通过 Slider 组件的 `onValueChange()` 方法就可以接收 Slider 滑动位置改变的消息，效果如图 4.8 所示。

需要提醒读者的是，Slider 组件的有些属性也是只支持特定平台的，例如 `minimumTrackTintColor` 和 `maximumTrackTintColor` 属性只支持 iOS 平台，所以在 Android App 中这两个属性没有生效，效果如图 4.9 所示。

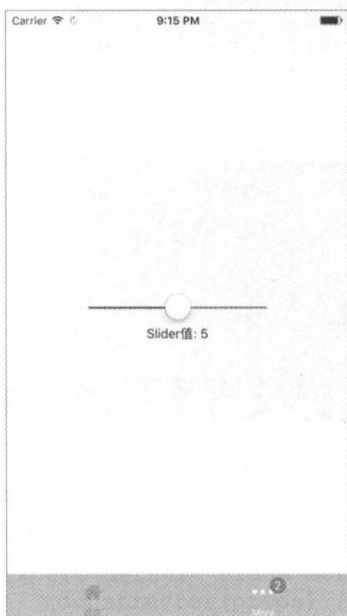


图 4.8 iOS 平台的 Slider 组件

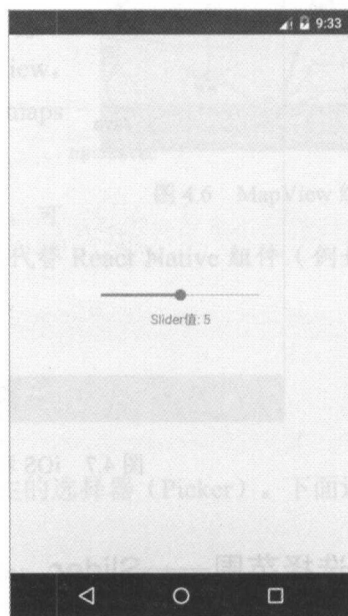


图 4.9 Android 平台的 Slider 组件

4.1.6 开关组件——Switch

Switch 是用来进行两个状态切换的组件，俗称开关组件，Switch 组件的用法也比较简单。但是有一点需要提醒读者的是：必须使用 `onValueChange()` 回调来更新 `value` 属性以响应用户的操作。如果不更新 `value` 属性，Switch 组件只会按一开始给定的 `value` 值来渲染且保持不变，看上去就像完全点不动的效果。

通过下面的代码来看看 Switch 的具体用法：

```

01 export default class more extends React.Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       isOn: false

```

```

06     }
07   }
08
09   render() {
10     return (
11       <View style={styles.container}>
12         <Switch onTintColor='blue'      // 开启时的背景颜色
13           thumbTintColor='green'      // 开关上原形按钮的颜色
14           tintColor='black'          // 关闭时背景颜色
15           onChange={() => this.setState({
16             isOn: !this.state.isOn
17           })} // 当状态值发生变化值回调
18           value={this.state.isOn === true} // 默认状态
19         />
20       </View>
21     );
22   }
23 }

```

Switch 组件的运行效果如图 4.10 所示。

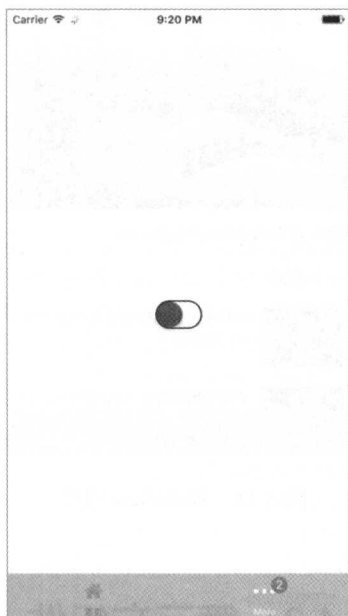


图 4.10 Switch 组件

4.1.7 打开网页——WebView

WebView 组件就像一个简单的手机浏览器，可以用来打开网页。例如，使用 WebView 打开网络地址 <https://sina.cn>，more.js 代码实现如下：

```

01 export default class more extends React.Component {
02   render() {
03     return (
04       <View style={styles.container}>
05         <WebView source={{
06           uri: 'https://sina.cn/'

```

```

07         }} style={styles.web}></WebView>
08     </View>
09     );
10 }
11 }
12
13 const styles = StyleSheet.create({
14     // 这里省略了没有修改的代码
15     web: {
16         width: 200,
17         height: 200
18     }
19 }

```

重新加载应用后，效果如图 4.11 所示。




图 4.11 WebView 组件

4.2 第三方组件

React Native 提供了大量的原生组件，但是为了进一步提升开发质量和效率，也可以使用第三方组件，例如，React Native 推荐使用 `react-native-maps` (<https://github.com/airbnb/react-native-maps>) 来代替原生组件 `MapView`。

那么如何找到这些优秀的第三方组件呢？这里推荐两个网站如下。

- GitHub (<https://github.com/>)：不仅包含 React Native 资源，还有大量其他平台开发用到的第三方组件和开源项目。
 - JS.COACH (<https://js.coach/react-native>)：主要是 JavaScript 资源，包含了 React Native。
- 对于本书的电商 App 已经开发的功能，又有哪些优秀的第三方组件可以替换呢？

 **提示：** 在开始使用第三方组件之前，这里先将 ch04 项目已有代码使用 git 进行版本控制，供本书后面的章节使用，具体方法可以使用 git tag 命令，即 git tag ch04-without-third-party-library。

4.2.1 react-native-swiper 的使用

对于轮播广告，除了使用 React Native 自带的 ScrollView 组件之外，就有一个优秀的第三方替代方案 react-native-swiper (<https://github.com/leecade/react-native-swiper>)。

(1) 将 react-native-swiper 添加到 ch04 项目中，安装的命令如下：

```
npm install react-native-swiper --save
```

(2) 使用 react-native-swiper 来替代原有基于 ScrollView 组件的轮播广告实现。修改 home.js 代码如下：

```
01 // 这里省略了没有修改的代码
02 import Swiper from 'react-native-swiper';
03
04 export default class home extends React.Component {
05   // 这里省略了没有修改的代码
06
07   render() {
08     return (
09       <View style={styles.container}>>
10         // 这里省略了没有修改的代码
11         <View style={styles.advertisement}>>
12           <Swiper loop={true} height={190} autoplay={true}>
13             {this.state.advertisements.map((advertisement,
14               index) => {
15               return (
16                 <TouchableHighlight key={index}
17                   onPress={() =>
18                     Alert.alert('你单击了轮播图', null,
19                       null)}>>
20                 <Image style={styles.advertisement
21                   Content}
22                   source={advertisement.image}>>
23                 </Image>
24                 </TouchableHighlight>
25               );
26             })}
27             </Swiper>
28           </View>
29           // 这里省略了没有修改的代码
30         </View>
31       );
32     }
33   }
```

(3) 重新加载应用，此时使用 react-native-swiper 实现的轮播广告，效果如图 4.12 所示。不过稍等片刻（大约 2 秒钟），应用却出现了错误，如图 4.13 所示。



图 4.12 使用 react-native-swiper 实现的轮播广告

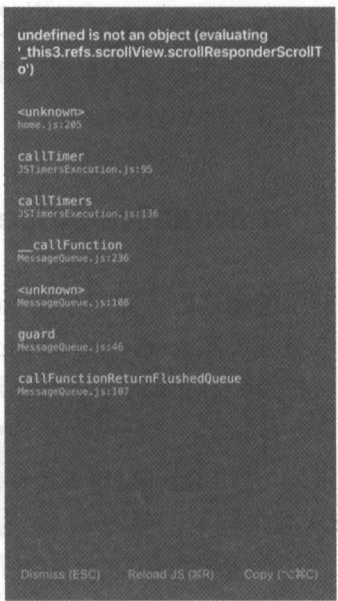


图 4.13 替代 ScrollView 后的错误

原来，当使用 react-native-swiper 实现轮播效果时，不需要再创建定时器来操作 ScrollView，只需要做简单的配置（`"loop={true} autoplay={true}"`）就可以达到效果。因此，要将 home.js 文件中 `_startTimer` 的相关代码全部删除。

4.2.2 NativeBase 的使用


NativeBase (<http://nativebase.io/>) 是一个优秀的 React Native 组件库，它同时被微软和 Awesome React Native 推荐，详见 <https://github.com/GeekyAnts/NativeBase>。当然，在读者使用过后或许会发现，这哪里仅仅是一个第三方组件，完全是一种要替代 React Native 原生 UI 组件的姿态。既然 NativeBase 这么强大，我们就赶快来体验一下吧。

(1) 首先要将 NativeBase 添加到 ch04 项目中，安装的命令如下：

```
npm install native-base --save
```


(2) 安装成功后，打开 package.json 文件，发现其中多了一条关于 NativeBase 的描述如下：

```
"native-base": "^0.5.20",
```

 **提示：** package.json 是 React Native 工程描述文件，该文件完整描述了 React Native 项目的信息和依赖。

(3) 由于 NativeBase 依赖于 react-native-vector-icons，所以还需要使用如下命令安装：

```
npm install react-native-vector-icons --save
react-native link react-native-vector-icons // 添加依赖 react-native-vector-
icons 到原生工程
```

 **注意：** 有时候在安装完新的第三方包后，会出现如图 4.14 所示的错误，此时解决办法是先停止 React Native 服务，然后再删除 node_modules 文件夹，接着使用 npm install 命令重新安装所有的依赖库，最后重新运行 React Native 服务和应用。

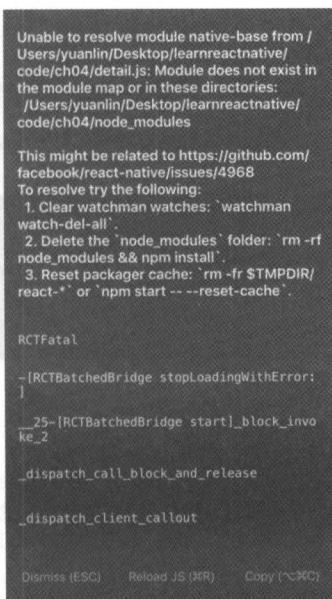



图 4.14 安装新的第三方包后发生的错误

下面就可以将原有的实现替换成 **NativeBase** 的相应组件了，这里以首页为例。

1. 首页的布局

按照 **NativeBase** 的 **Header** 和 **Content** 布局方式调整首页的布局结构：**NativeBase** 的所有组件都是放在 **Container** 组件中的，其中，**Header** 是导航栏组件，**Content** 组件用于实现页面正文。修改 **home.js** 代码如下：

```
01 // 这里省略了没有修改的代码
02 import {Container, Header, Content} from 'native-base';
03
04 export default class home extends Component {
05   // 这里省略了没有修改的代码
06
07   render() {
08     return (
09       <Container>
10         <Header>
11           <View style={styles.searchbar}>
12             // 这里省略了没有修改的代码
13           </View>
14         </Header>
15         <Content>
16           <View style={styles.advertisement}>
17             // 这里省略了没有修改的代码
18           </View>
19           <View style={styles.products}>
20             // 这里省略了没有修改的代码
21           </View>
22         </Content>
23       </Container>
24     );
25   }
26 }
```


 **提示：**在删除组件的同时，不要忘记删除相应的样式和布局代码。


重新加载应用后，基于 NativeBase 布局结构的应用运行效果如图 4.15 所示。



图 4.15 使用 NativeBase 布局结构的应用运行效果

2. 搜索框的样式

这时读者可能会发现：搜索框的样式有些问题。原来，NativeBase 提供了一套组件，这套组件的功能和样式是经过测试验证的，但是 NativeBase 组件与其他组件的兼容性需要调整，为了解决上述问题，最高效的办法就是使用 NativeBase 组件来实现搜索框，而非自己定义组件和布局。想要使用 NativeBase 实现搜索框功能，只需要为 Header 组件添加 searchBar 属性即可。修改 home.js 代码如下。

 **提示：**如果读者刚开始对 NativeBase 不熟悉的话，可以先阅读 NativeBase 的官方文档（<http://nativebase.io/docs/v0.5.13/>）来了解 NativeBase 组件的基本用法。

```
01 // 这里省略了没有修改的代码
02 import {Container, Header, Content, Button, InputGroup, Icon, Input}
   from 'native-base';
03
04 export default class home extends Component {
05   // 这里省略了没有修改的代码
06
07   render() {
08     return (
09       <Container>
10         <Header searchBar rounded> // Header 的配置: searchBar,
                                   rounded
11         <InputGroup>
12           <Icon name= 'ios-search-outline' />
                                   // 基于 react-native-vector-icons 的图标资源
```

```

13             <Input
14               placeholder='搜索商品'
15               onChangeText={(text) => {
16                 this.setState({searchText: text});
17                 console.log('输入的内容是 ' + this.state.
                  searchText);
18               }}/>
19             </InputGroup>
20             <Button
21               transparent
22               onPress={() => {
23                 Alert.alert('搜索内容 ' + this.state.searchText,
                  null, null);
24               }}>
25               搜索
26             </Button>
27           </Header>
28           // 这里省略了没有修改的代码
29         </Container>
30       );
31     }
32   }

```

⚠注意：上述代码中使用的 Button 组件是 NativeBase 中的 Button 组件，React Native 也有 Button 组件，所以在引入 NativeBase 中的 Button 组件时，需要删除 React Native 的 Button 组件，以避免命名冲突。

重载加载应用单击商品列表后，效果如图 4.16 所示。




图 4.16 使用 NativeBase 实现的搜索框

使用 NativeBase 实现搜索框，无须配置任何样式，但是显示的效果却如此好看！这就是使用 NativeBase 的强大之处：用更简单的代码达到更好的效果。NativeBase 组件已经实现了很多配置，开发者可以根据需求添加属性即可。例如，使用 `searchBar` 和 `rounded` 实现

导航栏的搜索框效果。

另外, NativeBase 使用的图标资源是基于 react-native-vector-icons (<https://github.com/oblador/react-native-vector-icons>) 项目的, 所以 Icon 组件使用资源只需要声明图标名称即可: name='ios-search-outline'。

 提示: 想要查询更多 react-native-vector-icon 项目可用的图标和名称, 可以通过该项目源代码或运行 Demo 例子查询。

3. 商品列表

NativeBase 提供了一系列的组件用于实现列表以及列表中图片文字的显示, 包括 List、ListItem、Thumbnail 以及 Text。修改 home.js 代码如下:

```
01 export default class home extends Component {
02   // 这里省略了没有修改的代码
03
04   render() {
05     return (
06       <Container>
07         <Header searchBar rounded>
08           // 这里省略了没有修改的代码
09         </Header>
10         <Content>
11           <Swiper loop={true} height={190} autoplay={true}>
12             // 这里省略了没有修改的代码
13           </Swiper>
14           <List>
15             <ListItem>
16               <Thumbnail
17                 square size={40}
18                 source={require('./images/product-
19                   image-01.jpg')} />
20               <Text>商品 1</Text>
21               <Text note>描述 1</Text>
22             </ListItem>
23           </List>
24         </Content>
25       </Container>
26     );
27   }
28
29   const styles = StyleSheet.create({ // 删除不用的组件样式
30     advertisementContent: {
31       width: Dimensions.get('window').width,
32       height: 180
33     }
34   });
```

重新加载应用, 效果如图 4.17 所示。



图 4.17 使用 NativeBase 实现的商品列表

同样，代码中没有做任何复杂的样式和布局的配置，就实现了类似之前商品列表的效果，而且还自动添加了分割线效果！

4. 数据和响应

添加 List 的数据源和单击响应。修改 home.js 代码如下：

```
01 export default class home extends Component {
02   constructor(props) {
03     super(props);
04     this.state = {
05       products: [
06         {
07           image: require('./images/advertisement-image-01.jpg'),
08           title: '商品 1',
09           subTitle: '描述 1'
10         }, {
11           image: require('./images/advertisement-image-01.jpg'),
12           title: '商品 2',
13           subTitle: '描述 2'
14         }, {
15           // 这里省略了重复的代码
16         }, {
17           image: require('./images/advertisement-image-01.jpg'),
18           title: '商品 10',
19           subTitle: '描述 10'
20         }
21       ],
22     }
23   }
24
25   render() {
26     return (
27       <Container>
28         <Header searchbar rounded>
```

```

29          // 这里省略了没有修改的代码
30      </Header>
31      <Content>
32          <Swiper loop={true} height={190} autoplay={true}>
33              // 这里省略了没有修改的代码
34          </Swiper>
35      <Content>
36          <List dataArray={this.state.products}
37              renderRow={this._renderRow}>
38              </List>
39          </Content>
40      </Container>
41  );
42  }
43
44  _renderRow = (product) => {
45      return (
46          <ListItem
47              button
48              onPress={() => {
49                  const {navigator} = this.props;
50                  if (navigator) {
51                      navigator.push({
52                          name: 'detail',
53                          component: Detail,
54                          params: {
55                              productTitle: rowData.title
56                          }
57                      });
58                  }
59              }}>
60              <Thumbnail square size={40} source={product.image}/>
61              <Text>{product.title}</Text>
62              <Text note>{product.subTitle}</Text>
63          </ListItem>
64      )
65  }
66  }

```

重新加载应用，使用 NativeBase 的 List 相关组件实现的商品列表效果如图 4.18 所示。

至此，一个全新的首页就完成了。对比使用 NativeBase 前后的 home.js 文件，可以发现这样一个令人震惊的事实：在实现相同功能的情况下，刚开始的 home.js 文件有 300 多行代码，而使用 NativeBase 后，home.js 文件代码行数约 150 行，只有之前的一半！而且 NativeBase 的样式和效果还比我们自己实现的更美观。因此，合理使用第三方组件，可以大大提高开发的效率和质量。



图 4.18 使用 NativeBase 实现的商品列表

4.2.3 NativeBase 如何解决跨平台问题

4.1 节为了实现分页的效果, 使用了平台差异化的方式: iOS App 使用的是 TabBarIOS 组件, 而 Android App 使用的是 ViewPagerAndroid 组件。如果使用 NativeBase 提供的 Footer Tabs 就可以解决这个问题。修改 main.ios.js 文件的代码如下:

```

01 import React, {Component} from 'react';
02 import {Container, Content, Footer, FooterTab, Badge, Button, Icon} from
'native-base';
03
04 import Home from './home';
05 import More from './more';
06
07 export default class main extends React.Component {
08   constructor(props) {
09     super(props);
10     this.state = {
11       selectedTab: 'home'
12     }
13   }
14
15   render() {
16     return (
17       <Container>
18         {this._renderContent()} // Content 的内容会随着 this.state.
                                selectedTab 值的变化而改变
19
20       <Footer >
21         <FooterTab>
22           <Button active={this.state.selectedTab === 'home'}
23             onPress={() => this.setState({selectedTab:
24               'home'})}>
25             首页
26           <Icon name='ios-apps-outline' />
27         </Button>
28         <Button active={this.state.selectedTab === 'more'}
29             onPress={() => this.setState({selectedTab:
30               'more'})}>
31           <Badge>2</Badge>
32           更多
33         <Icon name='ios-compass-outline' />
34       </Button>
35     </FooterTab>
36   </Content>
37 </Container>
38 );
39 }
40
41 _renderContent() {
42   if (this.state.selectedTab === 'home') {
43     return (

```

```
41         <Content>
42             <Home navigator={this.props.navigator}/>
43         </Content>
44     );
45     } else if (this.state.selectedTab === 'more') {
46         return (
47             <Content>
48                 <More navigator={this.props.navigator}/>
49             </Content>
50         );
51     }
52 }
```

重新加载 iOS 应用，效果如图 4.19 所示。
实现了 iOS App 的效果之后，再将 main.ios.js 文件的内容全部复制至 main.android.js 文件中。重新加载 Android 应用，效果如图 4.20 所示。

从 iOS App 和 Android App 的代码实现和运行效果来看，使用 NativeBase 的 Foot Tabs 组件又可以轻松地实现跨平台的代码复用：只需要重构下 ch04 项目的文件即可，修改 main.ios.js 文件名为 main.js，并且删除 main.android.js 文件。此时，ch04 项目结构如图 4.21 所示。



图 4.19 基于 NativeBase 实现的 iOS App 主页

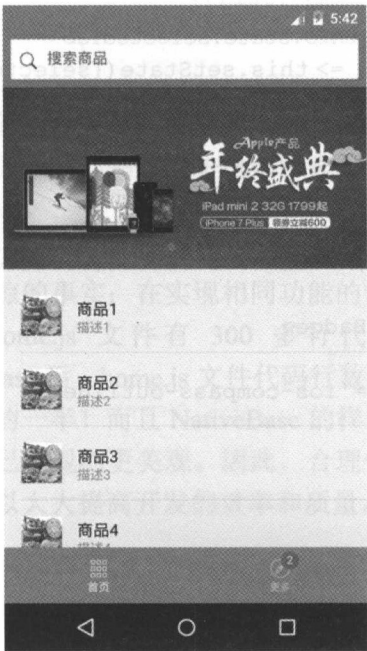


图 4.20 基于 NativeBase 实现的 Android App 主页

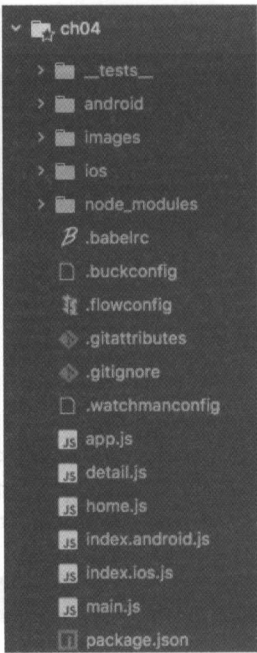


图 4.21 基于 NativeBase Foot Tabs 重构后的 ch04 项目

从上述使用第三方组件（包括 `react-native-swiper`、`NativeBase` 以及间接使用的 `react-native-vector-icons`）的例子中，想必读者已经感受到合理使用第三方组件，确实可以大大减少代码量、提高开发效率。但是，使用第三方组件也是有一定风险的，例如，作者停止维护、发现的问题修复不及时、组件的兼容性不好等。因此，在选择第三方组件的时候，需要仔细考察关注度、更新频率、引用次数以及开发者评价等因素。

4.3 小 结

高效工作一直是职场人士追求的目标，本章通过对一些特定平台组件和第三方组件的介绍，让读者能更高效地开发属于自己的 App。尤其是本书在电商 App 中引入了几个热门的第三方库，很大程度上简化了自己的代码实现，加快了应用开发的效率，大大降低了开发成本。

第 5 章 原生平台的适配和调试

使用 React Native 开发应用时，通常只需要了解 React Native 的相关知识即可，例如，本书介绍过的 React Native 调试、Flexbox 布局和适配、React Native 组件及第三方组件的使用。但是在一些情况下，了解一些原生平台的知识有助于快速定位和解决问题，如第 3 章完善首页的功能和样式时遇到的 ch04 没有注册的错误问题。所以，本章将为读者介绍一些平台适配问题。

本章主要内容有：

- iOS 平台的适配和调试。
- Android 平台的适配和调试。

5.1 iOS 平台的适配

使用 Xcode 打开 ch04 项目中 ios 文件夹下的 ch04.xcodeproj 文件，效果如图 5.1 所示。

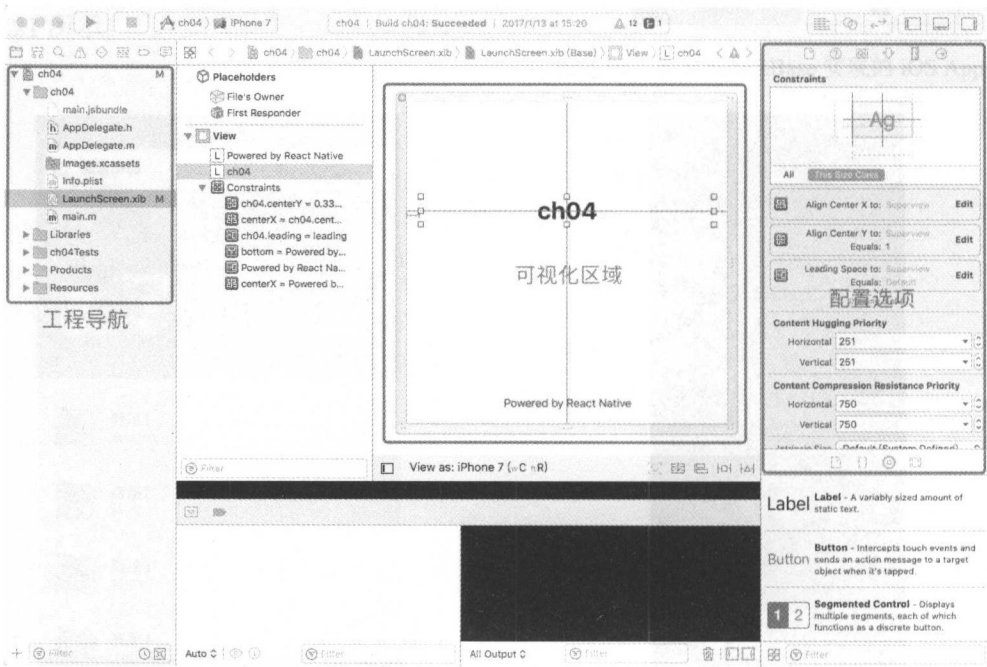


图 5.1 iOS 原生项目

除了使用源码方式实现适配之外，iOS 工程中用于适配的相关资源主要有 Images.xcassets、LaunchScreen.xib 以及工程中暂未使用的 storyboard 文件。

🔔 小知识：所有 iPhone 屏幕的详细尺寸和分辨率，可以参考该网站的汇总 The Ultimate Guide To iPhone Resolutions（<https://www.paintcodeapp.com/news/ultimate-guide-to-iphone-resolutions>）。

5.1.1 Images.xcassets 适配

Images.xcassets 已经为开发者定义好了所需资源的尺寸和用途，只需要按照说明添加响应的图片资源即可，如图 5.2 所示。

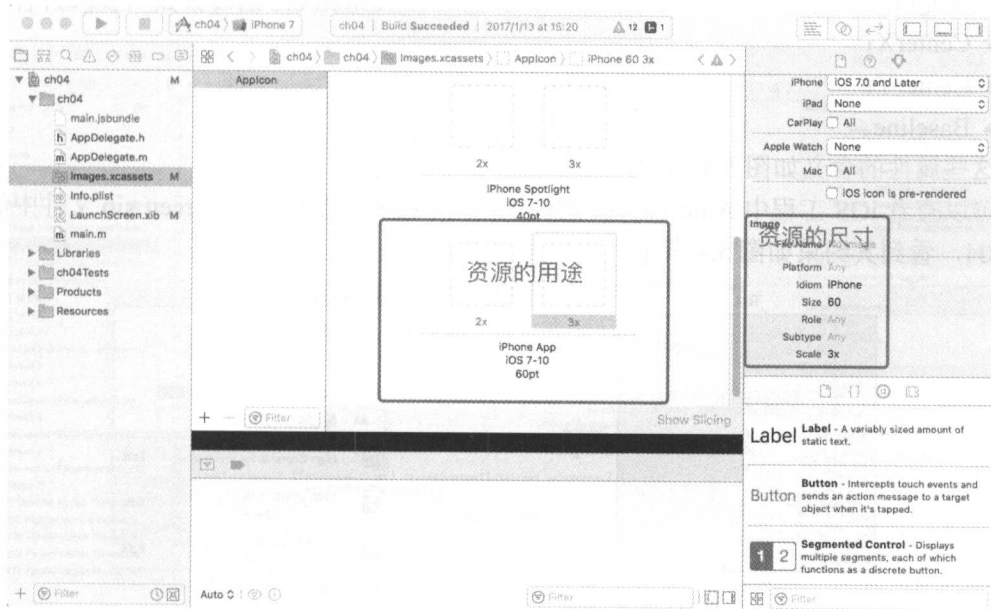


图 5.2 iOS 原生项目中的图片资源 Images.xcassets

例如，Images.xcassets 中的 AppIcon 指应用图标的资源，iPhone App iOS 7-10 60pt 表示在 iOS 系统版本 7~10 之间，尺寸为 60 个单位，分为 2x 和 3x 两种不同缩放比例，2x 用于 iPhone 6、iPhone 6s 和 iPhone 7 设备，图片的实际分辨率=2×60=120，3x 用于 iPhone 6 Plus、iPhone 6s Plus 及 iPhone 7 Plus 设备，图片的实际分辨率=3×60=180。

在 xib 或 storyboard 文件中，iOS 适配的技术基于自动布局（Auto Layout）系统，自动布局描述的是视图或组件之间的相对位置关系，它可以在应用运行时根据设备的屏幕尺寸动态改变各个视图或组件的尺寸。

5.1.2 自动布局 Auto Layout

在 React Native 开发中，想要实现类似 iOS 平台自动布局的方式，可以使用 Flexbox 布局，而 iOS 平台实现自动布局的方法是约束。约束既可以定义布局属性的具体值，也可以定位布局属性之间的关系。例如，可以为视图或组件添加如下约束：视图的高度是 4、两个视图之间的垂直距离是 10、这些视图的宽度相等。

常见的约束属性如下：

- Left;
- Right;
- Top;
- Bottom;
- Leading;
- Trailing;
- Width;
- Height;
- CenterX;
- CenterY;
- Baseline。

这些属性的含义如图 5.3 所示。

可以查看 iOS 工程中 Auto Layout 的例子，单击 iOS 工程 LaunchScreen.xib 文件中的标题 ch04，看到其约束如图 5.4 所示。

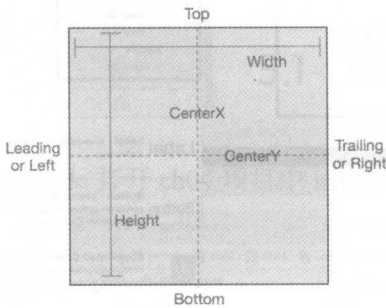


图 5.3 Auto Layout 的约束属性

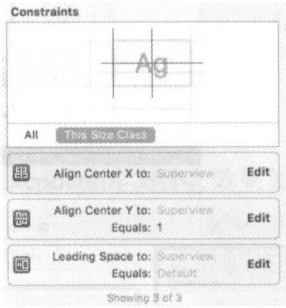


图 5.4 iOS 原生项目中标题 ch04 的约束

通过 CenterX 和 CenterY 设置该试图或组件相对于其 SuperView 位置的比例，就实现了不同尺寸屏幕的适配。

5.1.3 Size Class 适配

虽然 Auto Layout 解决了屏幕适配的问题，但是在实际开发过程中，不同设备的布局需求可能不完全相同。例如，在实现 iPhone 和 iPad 适配的时候，一个页面需要配置多个 xib 进行开发还是件很头疼的事情。好在从 iOS 8 开始，苹果引入了 Size Class。

使用 Size Class 之后，可以把各种尺寸屏幕的适配工作放在一个文件中完成，然后通过不同类别的 Size 来定制各种尺寸的界面。换句话说，xib 或 storyboard 文件不是一个普通的 xib 或 storyboard 文件，而是一个由多个不同布局合一的 xib 或 storyboard，可以管理多种类型的屏幕。Size Class 的使用如图 5.5 所示。

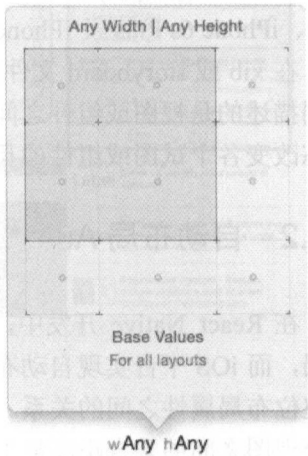


图 5.5 iOS 原生项目中的 Size Class

5.2 iOS 开发的调试技巧

在了解了 iOS 平台的适配方法之后，下面再来介绍一些 iOS 开发的调试技巧。

在刚才打开的 iOS 项目中，选择菜单 Product → Run 选项之后，和 react-native run-ios 命令效果一样，也可以运行 iOS App。在 iOS App 启动之前，单击 AppDelegate.m 文件中的代码行数 23，也可以添加一个断点，当 iOS App 启动后，应用会停止在刚才添加断点的第 23 行代码处，效果如图 5.6 所示。

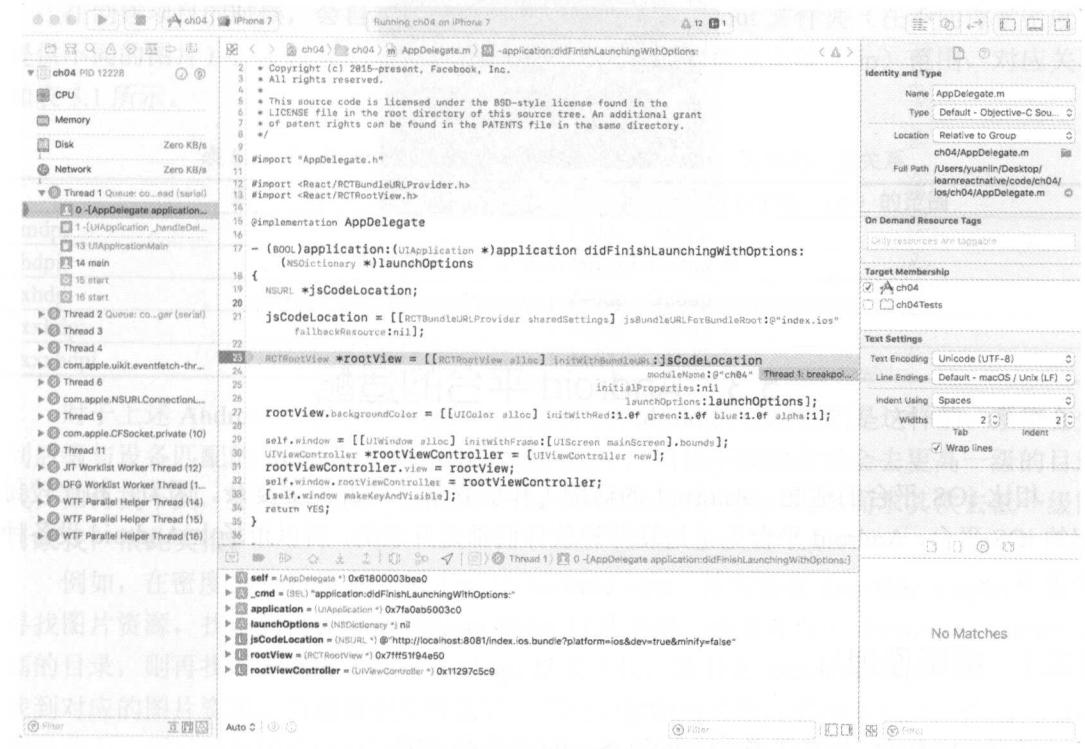


图 5.6 iOS 原生项目断点调试

和 React Native 中的断点调试一样，此时也可以查看当前应用运行的相关信息：线程信息及局部变量的值等。

当然，除了使用断点调试代码和逻辑外，iOS 平台还提供了 UI 调试的小工具，例如，打开 iOS 模拟器的菜单 Debug → Color Blended Layers 选项，就可以查看当前页面所有组件的位置和布局，效果如图 5.7 所示。

提示：关于 iOS 平台的 UI 调试，还有一些优秀的第三方工具，例如，FaceBook 出品开源免费的 chisel (<https://github.com/facebook/chisel>) 以及一款商业付费软件 Reveal (<https://revealapp.com/>)，它们可以在应用运行时动态地查看和调整 UI，从而大大提高布局开发的效率。

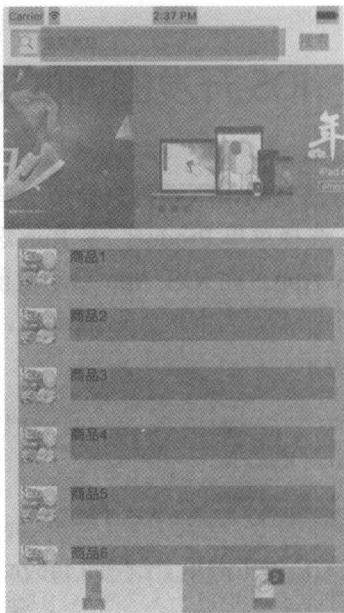


图 5.7 iOS App 的原生 UI 调试工具

5.3 Android 平台的适配

相比 iOS 平台的适配，Android 的适配工作更繁重，情况也更复杂。因为，相比较封闭的 iOS 平台，Android 平台无论是硬件和软件们都是开放的，所以市场上 Android 设备种类更多。

5.3.1 适配原理

打开 Android 开发工具 Android Studio，选择菜单 Open an existing Android Studio project 工具，打开 ch04 项目的 Android 文件夹，如图 5.8 所示。

通过 Android 的项目结构，想必读者可以隐约感受到：Android 适配是基于文件夹的，不同分辨率和尺寸的屏幕会自动适配相应文件夹下的布局或资源文件。但是，想要进一步理解 Android 的适配，有必要先了解 Android 适配的一些基本概念。

- 屏幕尺寸：屏幕尺寸是指手机屏幕对角线的英寸数。
- 屏幕分辨率：屏幕分辨率是指屏幕宽、高像素数。
- 屏幕像素密度：屏幕像素密度是指手机屏幕对角线上单位英寸内的像素数。

另外，编写代码时常用的尺寸单位如下。

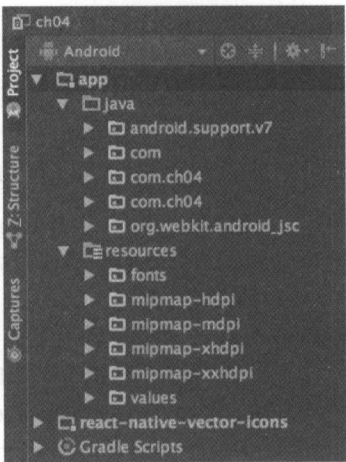


图 5.8 Android 原生项目结构

- px：像素。
- dp（dip 的缩写）：规定密度为 160 的屏幕上，1 像素对应的尺寸为 1dp。320 密度的屏幕上，1 像素对应 0.5dp，依此类推。在密度为 160 的屏幕上，1 英寸有 160 个像素，那么 1px 对应的尺寸=1/160 英寸，即 dp 是个物理尺寸，跟像素无关。所以，100dp 的尺寸在不同手机上显示出来，物理尺寸看上去基本是一样的。
- sp（Scale-independent Pixel），即与缩放无关的抽象像素。sp 和 dp 很类似，唯一的区别是，Android 系统允许用户自定义文字尺寸大小（小、正常、大、超大等），当文字尺寸是“正常”时，1sp=1dp=0.00625 英寸，而当文字尺寸是“大”或“超大”时，1sp>1dp=0.00625 英寸。

在创建项目的时候，会自动创建不同的 mipmap 或 layout 文件夹（在不同像素密度上提供不同的图片），文件夹的后缀表明了该布局或资源的像素密度（dp）范围，对应关系如表 5.1 所示。

表 5.1 Android 文件夹后缀命名与像素密度（dp）范围的对应关系

| 后 缀 | 像素密度（dp）的范围 |
|---------|-------------|
| mdpi | 120dp~160dp |
| hdpi | 160dp~240dp |
| xhdpi | 240dp~320dp |
| xxhdpi | 320dp~480dp |
| xxxhdpi | 480dp~640dp |

对于上述 Android 项目中的 mipmap 文件夹，Android 的适配机制是这样的：系统会先后缀与设备匹配的 mipmap 目录下找对应的图片，当找不到的时候会去更高一级的目录找，如再找不到，则继续往高一级的目录找，如果还是找不到，就退而求其次去低一级的目录找，依此类推。

例如，在密度为 xxhdpi 的手机上运行 Android App，首先会在 drawable-xxhdpi 目录下寻找图片资源，找不到再去 drawable-xxxhdpi 目录下找，如果没有比 drawable-xxxhdpi 更高的目录，则再找不到就去 drawable-xhdpi 目录下找，接着去 drawable-hdpi 目录下，直到找到对应的图片资源。当找到图片资源后，系统会按密度对图片做缩放处理，然后再显示到屏幕上。所以如果图片放的目录不正确的话，有可能造成图片因缩放而变得模糊。

同样，用于存放布局文件的 layout 目录也是通过后缀名来适配的，只不过 layout 文件夹通常添加设备分辨率作为后缀，如 layout-1280×720、layout-1920×1080 及 layout-land-1280×720 等。

不难看出，以上适配方法和上述 iOS 开发中的 Size Class 是类似的，即用于分类适配。

5.3.2 常用的适配属性

在 Android 实际编码中，为了支持不同屏幕尺寸，使用如下属性。

1. wrap_content方式

wrap_content 布局元素将根据内容更改大小，示例代码如下：

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```



```
03     android:layout_width="match_parent"
04     android:layout_height="match_parent"
05     android:orientation="vertical">
06
07     <Button
08         android:id="@+id/button1"
09         android:layout_width="wrap_content"
10         android:layout_height="wrap_content"
11         android:text="button1" />
12
13     <Button
14         android:id="@+id/button2"
15         android:layout_width="wrap_content"
16         android:layout_height="wrap_content"
17         android:text="button2" />
18
19 </LinearLayout>
```

此时，使用 `wrap_content` 方式布局的按钮效果如图 5.9 所示。

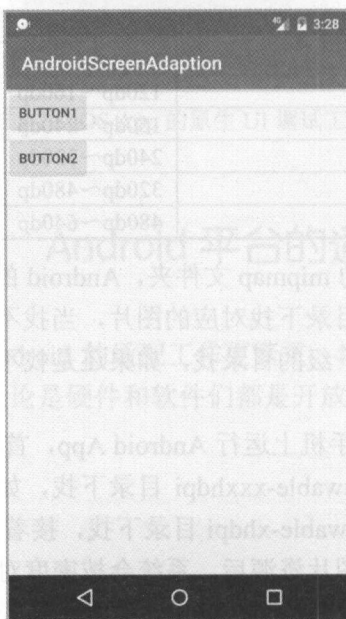


图 5.9 `wrap_content` 布局

2. `match_parent`方式

`match_parent` 自动填满整个父元素的空间，示例代码如下：

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03     android:layout_width="match_parent"
04     android:layout_height="match_parent"
05     android:orientation="vertical">
06
07     <Button
08         android:id="@+id/button1"
09         android:layout_width="match_parent"
10         android:layout_height="wrap_content"
```

```

11         android:text="button1" />
12
13     <Button
14         android:id="@+id/button2"
15         android:layout_width="match_parent"
16         android:layout_height="wrap_content"
17         android:text="button2" />
18
19 </LinearLayout>

```

此时，使用 `match_parent` 方式布局的按钮效果如图 5.10 所示。

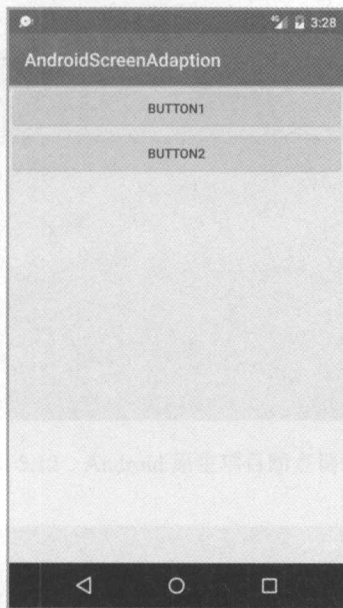



图 5.10 `match_parent` 布局

3. ConstraintLayout方式

尽管 Android 开发中可以使用上述布局方法,但是 Android 平台还是从 Android Studio 2.2 版本开始加入了一种全新的布局方法 `ConstraintLayout` (<https://developer.android.com/training/constraint-layout/index.html>)。简单来说,这种布局方式和上面介绍的 iOS 自动布局 (Auto Layout) 有相似之处,即描述的都是视图或组件之间的关系。

以下代码演示了如何使用 `ConstraintLayout` 方式实现子视图或组件居中的效果。

 **提示:** 如果想要体验全新的 `ConstraintLayout` 编辑器,请确保 Android Studio 的版本为 Android Studio 2.2 或更新。

```

01 <?xml version="1.0" encoding="utf-8"?>
02 <android.support.constraint.ConstraintLayout
03     xmlns:android="http://schemas.android.com/apk/res/android"
04     xmlns:app="http://schemas.android.com/apk/res-auto"
05     xmlns:tools="http://schemas.android.com/tools
06     android:id="@+id/activity_main"
07     android:layout_width="match_parent"
08     android:layout_height="match_parent"
09     tools:context="com.example.rn.androidscreenadaption.MainActivity">

```

```
10
11     <TextView
12         android:layout_width="wrap_content"
13         android:layout_height="wrap_content"
14         android:text="Hello World!"
15         app:layout_constraintBottom_toBottomOf="@+id/activity_main"
16         app:layout_constraintLeft_toLeftOf="@+id/activity_main"
17         app:layout_constraintRight_toRightOf="@+id/activity_main"
18         app:layout_constraintTop_toTopOf="@+id/activity_main" />
19
20 </android.support.constraint.ConstraintLayout>
```

此时，使用 `ConstraintLayout` 方式布局的按钮效果如图 5.11 所示。

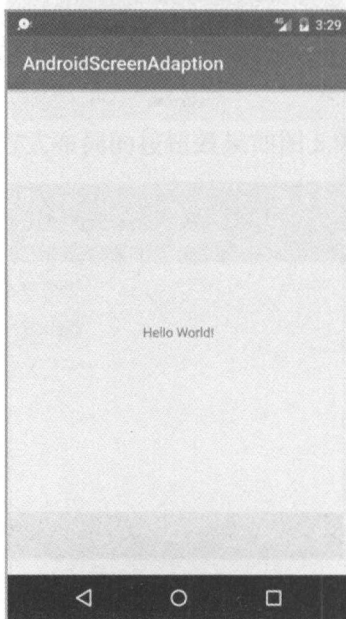


图 5.11 `ConstraintLayout` 布局

5.4 Android 平台的调试技巧

Android 平台的调试和其他平台的调试也很类似，例如，在 `Android Studio` 打开的工程中，打开源码 `MainActivity.java`，然后将鼠标光标移至代码编辑区的左侧后，单击即可添加断点，接着选择菜单 `Debug` → `Debug 'app'` 选项，即可调试 Android 应用，效果如图 5.12 所示。

如果要调试原生项目的实现和逻辑，可以使用上述方法；如果要调试 Android 的布局，可以使用 `React Native` 的调试选项 `Toggle Inspector`。

晃动 Android 设备或者在 Android 模拟器上使用快捷键（`command + M`）打开调试选项，然后选择 `Toggle Inspector` 选项，此时单击任意组件，即可查看该组件的布局信息，效果如图 5.13 所示。

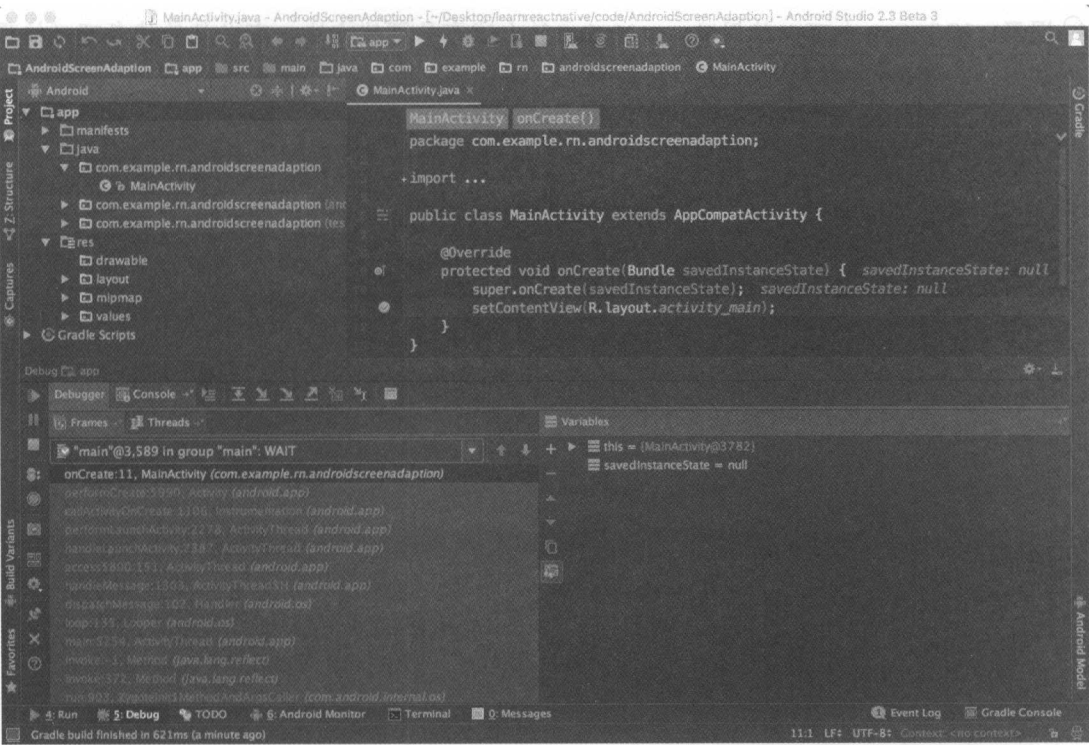


图 5.12 Android 原生项目断点调试

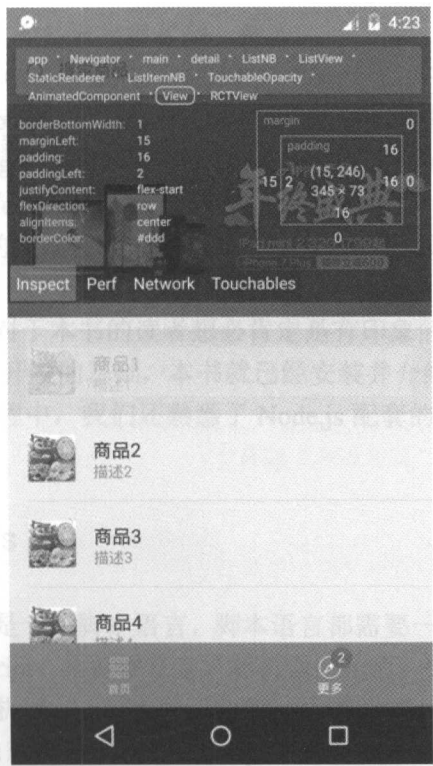



图 5.13 打开 Toggle Inspector 调试布局

提示: iOS App 也可以使用 React Native 的上述布局调试方法, 有兴趣的读者可以按照 Android App 的介绍自己尝试一下。

5.5 小 结

平台的适配一直是创始人或 CTO 最关心的问题, 这涉及开发成本和开发周期, 所以本章的内容对创业者很关键。笔者殷切地期望通过本章的学习和讨论后, 读者能够做到独立地开发基于 React Native 的应用, 学而时习之, 自己动手练习和编码才是掌握一门开发技能的最有效方法。

第 6 章 React Native 的服务器端处理

在本书的 2~4 章中，我们已经实现了电商应用的主要功能。但是除了修改代码的方法之外，应用内的数据是无法更新的，而应用的实际数据往往都是从服务器动态获取的。因此，本章将扩展一个新的开发角度：结合服务端来完善我们的 React Native 应用。

本章主要内容有：

- 掌握 Node.js 的原理和开发流程。
- 学习服务器端接口设计规范 RESTful。
- 了解网络前后端交互的原理。
- 实现 App 从服务器获取数据。
- 实现 App 数据的本地化存储。

6.1 学 习 Node.js

常见的服务端开发语言和技术有很多种，例如：

- 基于 Java 的 Spring (<https://spring.io/>)；
- 基于 Ruby 的 Ruby on Rails (<http://rubyonrails.org/>)；
- 基于 Python 的 Django (<https://www.djangoproject.com/>)；
- 基于 JavaScript 的 Node.js (<https://nodejs.org/en/>)；
- 以及其他一些常用的建站技术，例如 PHP (<http://www.php.net/>)、ASP (<https://www.asp.net/>) 等。

但是一说到 Node.js，对于本书的读者想必肯定是有印象的，在搭建 React Native 开发环境中，作为 React Native 开发的基础，本书就已经安装并介绍了 Node.js 的使用，并且在 React Native 应用的开发过程中，我们还熟悉了 Node.js 配套的 npm（Node.js 的包管理器）工具的使用。

6.1.1 什么是 Node.js

众所周知，JavaScript 是一门脚本语言，脚本语言都需要一个解析器才能运行。对于写在 HTML 页面里的 JavaScript，浏览器充当了解析器的角色。而对于独立运行的 JavaScript 代码，Node.js 就是它的解析器。


所以简单来说，Node.js 就是一个让 JavaScript 运行在服务端的开发平台，它让 JavaScript 成为脚本语言世界的一等公民，在服务端堪与 Ruby、Python、PHP 平起平坐。Node.js 基于 Google V8 JavaScript 引擎，V8 引擎执行 JavaScript 的速度非常快，性能非常好。

问题:都说 Node.js 是用来开发服务端程序的,可是开发者往往使用 Node.js 开发 React Native 移动端应用啊?

回答:这里所说的服务端,指的是广义上的 C-S (Client-Server) 架构,使用 React Native 开发应用时,首先也会运行一个 Node.js 服务来监听和响应应用的请求,所以这种说法并不矛盾。

6.1.2 为什么选择 Node.js

虽然了解了 Node.js 是什么,却仍然不能解决这样的疑问:为什么选择 Node.js? 或者说 Node.js 都有哪些优势呢?

 **提示:**为了表述简便,同时遵守官方的命名规范,本书下文中的 Node.js 简称 Node,请读者知悉。

1. 统一的开发语言

Node 是使用 JavaScript 语言开发的,而读者已经知道 JavaScript 不仅可以开发 Web 前端以及 React Native 移动端,还能够开发服务端,这样 JavaScript 就变成了一个前后端“通吃”的语言。对于开发者来说,无疑是一个巨大的福音:掌握一门开发语言,就可以开发不同平台不同场景的程序。

2. 简单易学

从开发语言的角度来看,JavaScript 相比 C、C++ 以及 Java 等语言入门简单很多,如果开发者已经了解 Web 前端或 React Native 开发,那么上手 Node 将更加容易。

从原理上看,Node 又保持了 JavaScript 在浏览器中单线程的特点。单线程的最大好处是不用像多线程编程那样过多地考虑同步问题,这里没有死锁的存在,也没有线程上下文交换所带来的性能上的开销。

因此在 Node 中,绝大多数的操作都以异步的方式进行调用,例如下面读取文件内容的代码:

```
01 const fs = require('fs');
02
03 fs.readFile('/path/to/file1', function (err, file) {
04     console.log('读取文件 1 完成');
05 });
06
07 fs.readFile('/path/to/file2', function (err, file) {
08     console.log('读取文件 2 完成');
09 });
10
11 console.log('读取文件 1 和文件 2');
```

由于 Node 的异步特性,上述代码打印信息如下:


```
读取文件 1 和文件 2
读取文件 1 完成
读取文件 2 完成
```


如果文件 2 早于文件 1 读取完毕的话，打印信息还可能是：

读取文件 1 和文件 2
读取文件 2 完成
读取文件 1 完成

3. 高性能

Node 使用异步 I/O 和事件驱动代替多线程，带来了很大的性能提升。另外，Node 在使用强大的 Google V8 JavaScript 引擎的同时，还使用了高效的 libev 和 libeio 库支持事件驱动和异步 I/O。

 **小知识：**Google V8 JavaScript 引擎是由 Google 开发的开源 JavaScript 引擎，最开始主要用于 Google Chrome 中。Google 研发 V8 引擎的主要原因是因为 Google 对既有 JavaScript 引擎的执行速度不满意。因此，大幅提升 JavaScript 执行效率和性能的 V8 引擎一经推出，就由于良好的性能吸引了相当的注目。

4. 跨平台

从本书搭建 React Native 开发环境中 Node 的安装过程中读者可能知道：Node 安装包支持现在主流的几乎所有的操作系统，如 Windows、macOS、Linux，同时，还可以下载源代码自行编译安装。

另外，Node 底层核心模块主要由 C/C++ 语言编写，因此，很容易做到跨平台。Node 基于 libuv 实现的跨平台架构如图 6.1 所示。

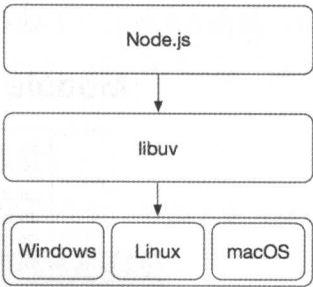


图 6.1 Node 跨平台结构

5. 社区活跃

Node 有着非常庞大的开发者社区和很高的活跃度，相比 React Native 的生态系统，Node 项目在 Github 上的关注度和更新频率也非常高。两者对比如图 6.2 和图 6.3 所示。

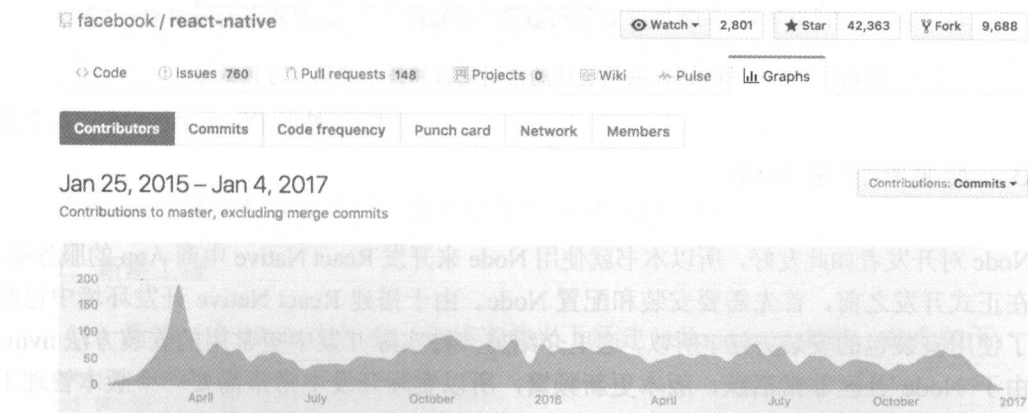


图 6.2 React Native 项目在 Github 上的关注度和贡献

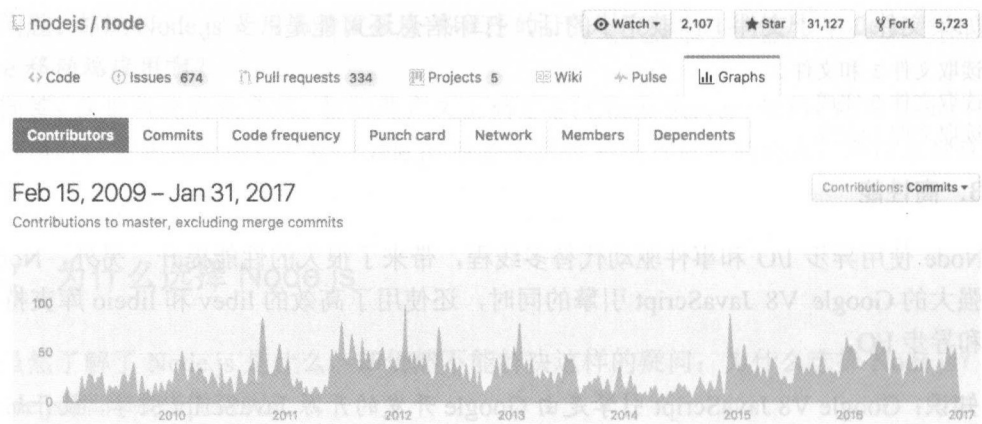


图 6.3 Node 项目在 Github 上的关注度和贡献

同时，围绕 npm 建立起来的庞大的第三方包生态圈，大大提高了 Node 开发的效率。网站 Module Counts (<http://www.modulecounts.com/>) 对比了常用语言 and 技术的第三方包数量，如图 6.4 所示。不难看出，npm 的第三方包数量遥遥领先！

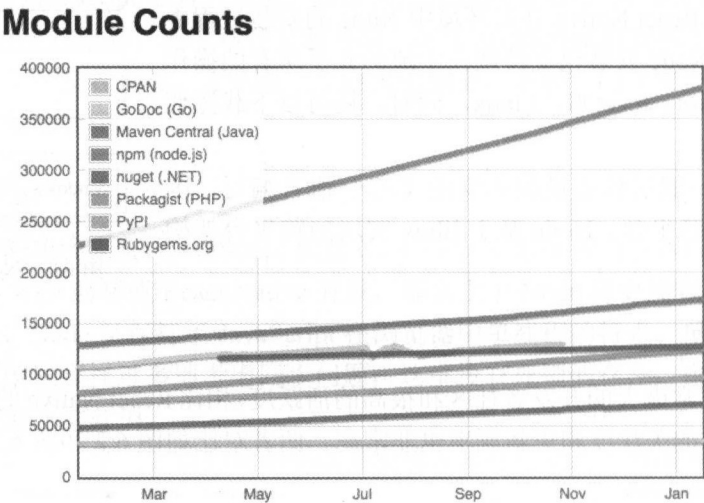


图 6.4 Module Counts 网站统计的常用语言 and 技术的第三方包数量

6.1.3 安装和使用 nvm

Node 对开发者如此友好，所以本书就使用 Node 来开发 React Native 电商 App 的服务端。在正式开发之前，首先需要安装和配置 Node。由于搭建 React Native 开发环境中已经介绍了使用安装包的安装方法。所以，这里介绍另一种实际开发中更常用的安装方法 nvm。由于 Node 社区非常活跃，版本更新频繁，所以实际开发中常常需要一个版本管理工具，nvm (<https://github.com/creationix/nvm>) 就是这样一个 Node 版本管理器。

提示：除了 nvm 之外，Node 的版本管理器还有 n (<https://github.com/tj/n>)，没错，名字就是这么简单。


(1) 在 macOS 或 Linux 系统上, nvm 的安装比较简单, 使用如下命令即可。

```
// 使用 curl
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.0/install.sh | bash
```

```
// 或者使用 wget
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.33.0/install.sh | bash
```

在 Windows 系统上, 可以下载安装包来安装, 项目地址为 <https://github.com/coreybutler/nvm-windows>。

(2) nvm 安装成功后, 可以通过如图 6.5 所示的命令进行验证。

 提示: 如果 nvm 安装成功之后, 还是找不到 nvm 工具或命令, 可以尝试重新加载环境变量或重启终端。

(3) 使用如下命令查看和安装 Node。

```
nvm ls-remote          // 查看可用的 Node 版本列表
nvm install 6.9.0       // 这里安装 LTS 版本 6.9.0
```

(4) 安装完 LTS 版本的 Node 之后, 将该版本 Node 设置成默认使用的 Node, 命令如下:

```
nvm alias default 6.9.0 // 设置 6.9.0 版本为默认使用的 Node
```

(5) 最后, 可以通过如下命令验证 Node 安装和 nvm 配置是否成功, 如图 6.6 所示。

```
+ ~ nvm --version
0.31.1
+ ~
```

图 6.5 查看 nvm 版本号

```
+ ~ nvm ls
->      v6.9.0
      system
default -> 6.9.0 (-> v6.9.0)
node -> stable (-> v6.9.0) (default)
stable -> 6.9 (-> v6.9.0) (default)
iojs -> N/A (default)
+ ~ node --version
v6.9.0
+ ~
```

图 6.6 查看 nvm 安装的 Node 版本和当前使用的 Node 版本


6.1.4 Node.js 的开发流程

在 Node 安装和配置成功后, 就可以使用 Node 进行开发了。

1. 新建工程

(1) 新建一个 Node 文件夹, 在该文件夹下新建 hellonode.js 文件, 命令如下:

```
mkdir Node
cd Node
touch hellonode.js
```

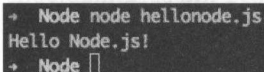
 提示: 上述命令的用法和解释, 读者可以参考第 2 章的内容。

(2) 在 `hellonode.js` 文件中添加如下代码:

```
01 console.log("Hello Node.js!"); // 打印 "Hello Node.js!"
```

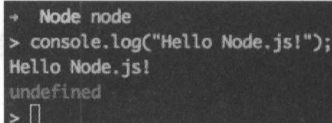
(3) 此时, 使用 `node` 命令执行该文件, 效果如图 6.7 所示。

当然, 除了使用文件的方式运行 `Node` 程序外, 对于简单程序, 还可以使用命令行交互的方式, 如图 6.8 所示。



```
+ Node node hellonode.js
Hello Node.js!
+ Node
```

图 6.7 使用 `node` 执行 `hellonode.js` 文件



```
+ Node node
> console.log("Hello Node.js!");
Hello Node.js!
undefined
>
```

图 6.8 `Node` 命令行模式

(4) 除了上述打印日志的基本功能外, 再看看 `Node` 读取文件的实现, 来体验一下 `Node` 异步 I/O 操作。新建两个测试文件 `file1` 和 `file2`, `file1` 的内容是“文件 1”, `file2` 的内容是“文件 2”:

```
echo 文件 1 > file1
echo 文件 2 > file2
```

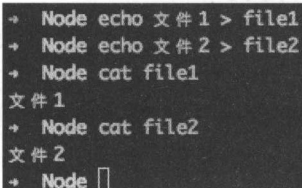
提示: 上述命令中, `echo` 用来将一段字符输出到标准输出上, “>”是重定向符, 表示将 `echo` 命令输出的字符重定向到名为 `file1` 的文件中。当然, 读者也可以使用自己熟悉的代码编辑器实现同样的效果。

效果如图 6.9 所示。

(5) 新建 `readfile.js` 文件, 并添加以下代码:

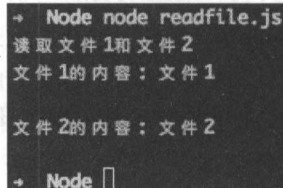
```
01 const fs = require('fs');
02
03 fs.readFile('file1', function (err, file) {
04     console.log('文件 1 的内容: ' + file);
05 });
06
07 fs.readFile('file2', function (err, file) {
08     console.log('文件 2 的内容: ' + file);
09 });
10
11 console.log('读取文件 1 和文件 2');
```

(6) 此时, 使用 `node` 命令执行该文件, 结果如图 6.10 所示。



```
+ Node echo 文件 1 > file1
+ Node echo 文件 2 > file2
+ Node cat file1
文件 1
+ Node cat file2
文件 2
+ Node
```

图 6.9 新建测试文件



```
+ Node node readfile.js
读取文件 1和文件 2
文件 1 的内容: 文件 1
文件 2 的内容: 文件 2
+ Node
```

图 6.10 使用 `node` 执行 `readfile.js` 文件结果

从这里的打印信息可以看出: `Node` 不需要等待文件读取完, 就会在读取文件时同时执

行下面的代码，从而大大提高了程序的性能。另外，由于异步操作是非阻塞的，所以 Node 会使用回调函数来传递文件读取的结果。

2. HTTP服务器

可能读者对于上面的例子觉得还是太简单了，因为打印日志和读取文件，对于任何语言都“不在话下”。而 Node 可以说是为网络开发而生的平台，所以能做到的事情当然远不止这些，下面使用 Node 开发一个服务端程序。

有过 Java 或 PHP 开发经验的读者可能知道：当使用 Java 或 PHP 来开发服务器时，需要 Apache (<http://httpd.apache.org/>) 或 Nginx (<http://nginx.org/>) 的 HTTP 服务器，即“浏览器 - HTTP 服务器 - PHP 解释器”的结构。但是对于 Node 来说，它不仅仅实现了应用，同时还实现了 HTTP 服务器，即 Node 将 HTTP 服务器剥离开，让服务端的结构简化成“浏览器 - Node”。在进一步了解了使用 Node 进行服务端开发的优势之后，赶紧开始下面的例子吧。

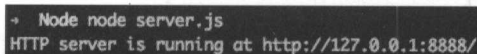
(1) 新建 server.js 文件，并添加如下代码：

```
01 // 载入 HTTP 模块
02 const fs = require('fs');
03
04 http.createServer(function (req, res) {
05     // HTTP 状态值: 200, 内容类型: text/html
06     res.writeHead(200, {'Content-Type': 'text/html'});
07     res.write('<h1>Hello Node.js!</h1>');
08     res.end('<p>The End</p>');
09 }).listen(8888);
10
11 console.log("HTTP server is running at http://127.0.0.1:8888/");
```

(2) 仍然使用 node 命令执行该文件，效果如图 6.11 所示。

 **注意：**此时程序并没有结束退出，通过 createServer 创建的 server，调用 listen() 方法会一直监听指定的端口号 8888。

(3) 打开浏览器访问地址 <http://127.0.0.1:8888/>，就会看到内容为 Hello Node.js! 的网页，如图 6.12 所示。



```
* Node node server.js
HTTP server is running at http://127.0.0.1:8888/
```

图 6.11 使用 node 执行 server.js 文件结果

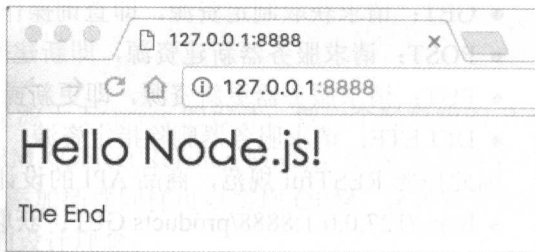


图 6.12 使用浏览器请求 Node 服务器

至此，用 Node 实现的最简单的 HTTP 服务器就完成了。

6.2 服务端接口的设计：RESTful

在掌握了 Node 的基本概念和用法之后，原本就可以为 React Native 电商 App 开发服务端程序了。但是本着先理清设计再开发的原则，有必要先了解服务端接口的设计规范 RESTful API。


对于 RESTful API 规范，首先令人不解的应该就是 REST 的含义了。

REST (Representational State Transfer 的缩写) 即表述性状态转移，是 Roy Fielding 博士 (https://en.wikipedia.org/wiki/Roy_Fielding) 在其 2000 年的博士论文中提出来的一种软件架构风格。表述性状态转移是一组架构约束条件和原则，满足这些约束条件和原则的应用程序或设计就是 RESTful。

不过，概念远没有例子更容易理解，所以这里以 RESTful 定义的电商应用 API 为例，来让读者对 RESTful 有一个更直观的认识。

按照 RESTful 的设计原则即每一个网址代表一种资源，所以地址中一般只是用名词，而不用动词，例如

- <http://127.0.0.1:8888/advertisements>: 表示广告 API。
- <http://127.0.0.1:8888/products>: 表示商品 API。

 **提示：**上述 RESTful API 的服务器地址，在实际开发中要替换成实际服务器地址。

而对资源的不同操作，通过 HTTP 协议定义的方法来区分。其中，HTTP 协议定义的常用方法如下。

- GET: 请求获取指定资源。
- POST: 向指定资源提交数据。
- PUT: 请求服务器存储一个资源。
- DELETE: 请求服务器删除指定资源。
- HEAD: 请求指定资源的响应头。
- OPTIONS: 返回服务器支持的 HTTP 请求方法。

RESTful API 主要使用以下 4 种 HTTP 方法操作资源。

- GET: 请求获取指定资源，即查询操作。
- POST: 请求服务器新建资源，即新建操作。
- PUT: 请求服务器更新资源，即更新操作。
- DELETE: 请求服务器删除指定资源，即删除操作。

因此按照 RESTful 规范，商品 API 的设计如下。

- <http://127.0.0.1:8888/products> GET: 获取所有商品。
- <http://127.0.0.1:8888/products> POST: 新建商品。
- <http://127.0.0.1:8888/products/ID> PUT: 更新某一指定 ID 的商品。
- <http://127.0.0.1:8888/products/ID> DELETE: 删除某一指定 ID 的商品。

如果查询操作获取的数量太多，还可以在 API 中添加过滤信息。

- <http://127.0.0.1:8888/products?limit=10> GET 为获取指定数量的商品。

- <http://127.0.0.1:8888/products?offset=10> GET 为获取指定开始位置的商品。
- http://127.0.0.1:8888/products?page=2&per_page=10 GET 为获取指定页面和数量的商品。
- http://127.0.0.1:8888/products?product_type=1 GET 为获取指定类型的商品。

当然，以上只是对 RESTful 规范做了一个简单的介绍，实际开发中还需要考虑 Web 页面和 API 的分离、API 版本控制以及 HTTP 状态码等。但是上述内容已经基本满足本书的开发需求，所以在此就不做过多介绍了。

想要了解更多 RESTful 规范，读者可参考相关书籍和教程。

6.3 实现电商 App 的服务器端接口

在熟悉了 RESTful API 的设计之后，就可以开始动手开发我们的服务器了。

从上面的例子可以知道，Node 已经提供了 HTTP 模块，但是在实际开发中，却不会直接使用它来进行 Web 开发，因为 Node 提供的 HTTP 模块仅仅是对 HTTP 服务器内核进行了封装，但是实际开发的 Web 服务器，不仅仅需要处理 HTTP 请求，还包括 Cookie 和会话管理、路由控制以及模板渲染等。因此，可以使用第三方 Web 框架来加快服务器开发。

6.3.1 Express 框架


这里笔者选择 Express (<http://www.expressjs.com.cn/>) 开发框架，因为它是目前最稳定、功能最强大而且使用也最广泛的 Node 框架。

 小知识：比较流行的 Node 开发框架还有 Koa (<http://koajs.com/>)。

Express 除了为 HTTP 模块提供了封装之外，还实现了 Web 开发中常用的如下功能：

- 用户会话；
- 路由控制；
- 模板解析；
- 静态文件服务；
- 错误控制器；
- 访问日志；
- 缓存；
- 插件。

这里需要重点关注的是插件的支持。这是源于 Express 的设计：做一个轻量级的 Web 框架。例如，Express 并不支持例如常见的 ORM (Object Relation Model, 对象关系模型)，但是 Express 支持并拥有大量的第三方插件，添加插件同样可以实现 ORM。这种插件化的设计，大大降低了耦合性，是一种值得借鉴的设计理念。

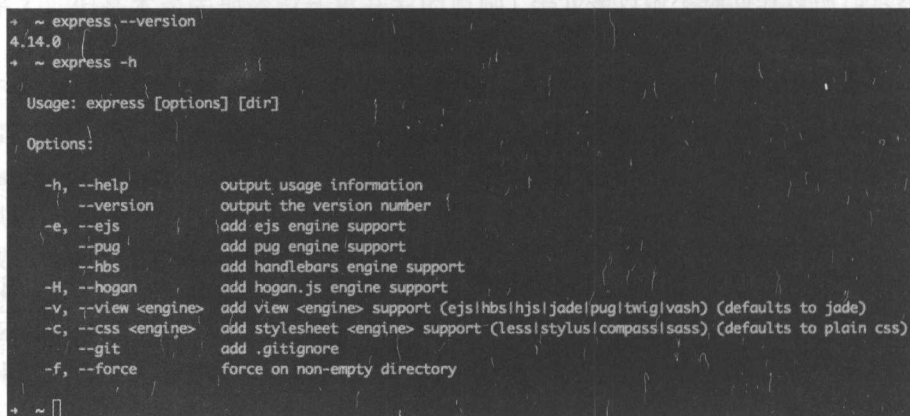
 小知识：耦合性 (Coupling) 也叫耦合度，是对模块间关联程度的度量。模块间的耦合度是指模块之间的依赖关系，包括控制关系、调用关系、数据传递关系。模块间联系越多，其耦合性越强，同时表明其独立性越差。软件设计中通常用耦合度和内聚度作为衡量模块独立程度的标准。划分模块的一个准则就是高内聚低耦合。

1. 安装和使用Express

(1) 安装 Express 应用生成器，命令如下：

```
npm install express-generator -g // -g 参数表示全局安装
```

(2) 安装成功后，可以通过如图 6.13 所示方法验证。



```
* ~ express --version
4.14.0
* ~ express -h


Usage: express [options] [dir]

Options:

  -h, --help            output usage information
      --version          output the version number
  -e, --ejs             add ejs engine support
      --pug             add pug engine support
      --hbs             add handlebars engine support
  -H, --hogan           add hogan.js engine support
  -v, --view <engine>  add view <engine> support (ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)
  -c, --css <engine>   add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)
      --git             add .gitignore
  -f, --force          force on non-empty directory

* ~
```

图 6.13 查看 Express 版本号和帮助

 **提示：**如果不是从头新建 Express 项目，而是想要在已有的 Node 项目中添加 Express，可以使用 `npm install express --save` 命令，但是使用 Express 应用生成器的更大好处是自动生成了项目的模板代码。

(3) 好了，现在就可以使用 `express` 命令快速创建服务端项目了，命令如下：

```
express --ejs Server // 新建 Server 项目，并使用 ejs 模板引擎
```

此时，`express` 命令会自动新建 `Server` 项目，并且生成如下文件：

```
create : Server
create : Server/package.json
create : Server/app.js
create : Server/public
create : Server/public/images
create : Server/public/javascripts
create : Server/routes
create : Server/routes/index.js
create : Server/routes/users.js
create : Server/public/stylesheet
create : Server/public/stylesheet/style.css
create : Server/views
create : Server/views/index.ejs
create : Server/views/error.ejs
create : Server/bin
create : Server/bin/www
```

项目成功创建后，还会提示如何安装依赖和运行项目，如图 6.14 所示。

(4) 按照提示安装依赖，命令如下：

```
cd Server
npm install
```

(5) 安装完依赖包后, 就可以运行基于 Express 框架的服务器了, 命令如下:

```
npm start
```

此时, 效果如图 6.15 所示。

```
install dependencies:
$ cd Server && npm install

run the app:
$ DEBUG=server:* npm start
```


图 6.14 提示如何安装依赖和运行项目

```
+ Server npm start
> server@0.0.0 start /Users/.../Server
> node ./bin/www
```

图 6.15 运行 Server 项目

 提示: 和 6.1.4 节 Node.js 开发流程中使用 HTTP 模块创建的服务器例子一样, 这里也使用了 npm start 命令。

(6) 打开浏览器访问地址 <http://127.0.0.1:3000/>, 就可以看到 Express 页面了, 如图 6.16 所示。

 提示: 使用 Express 应用生成器生成的项目, 默认使用 3000 端口, 如果想要修改端口号的话, 可以在运行项目的时候添加 PORT 参数 PORT=8888 npm start, 重新运行项目, 此时监听的端口号就是指定的端口 8888, 效果如图 6.17 所示。

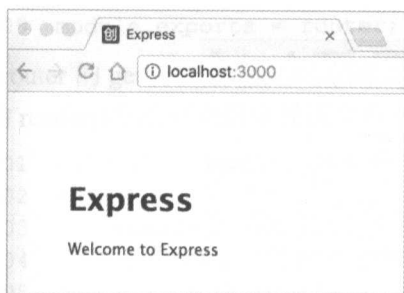


图 6.16 访问 Server 项目

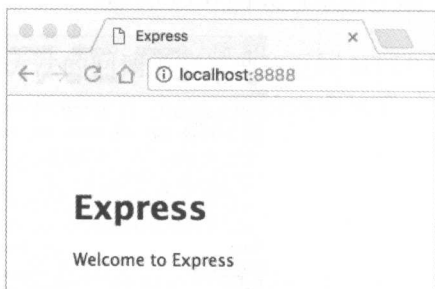



图 6.17 配置 Server 项目端口号

2. Express 项目结构

现在回头审视下刚才新建的 Server 项目。使用 Atom 编辑器打开 Server 项目, 效果如图 6.18 所示。

 提示: Node 开发常用的编辑器有 WebStorm (<https://www.jetbrains.com/webstorm/>)、Visual Studio Code (<https://code.visualstudio.com/>)、Atom (<https://atom.io/>) 以及 Sublime Text (<https://www.sublimetext.com/>) 等, 读者可以根据自己的喜好选择习惯的编辑器开发 Node。

其中, 各文件夹的作用和说明如表 6.1 所示。



图 6.18 Server 项目结构

表 6.1 Express目录和文件说明

| 目录/文件 | 说 明 |
|--------------|---|
| bin | 可执行文件，用于配置和启动工程入口文件 |
| public | 静态资源目录 |
| routes | 路由目录 |
| views | 模板文件，这里使用的是ejs模板引擎 |
| app.js | 入口文件 |
| package.json | 工程配置文件，描述了工程的所有信息以及第三方库的依赖关系，例如，刚才初始化的Server工程依赖的Express版本为4.14.0 |

由于，服务端主要是为 React Native 应用提供 API，所以这里暂不需要了解与页面渲染相关的目录或文件，例如 views 文件夹，本节开发过程中主要涉及的目录或文件如下。

- public 目录：用于存放图片资源。
- routes 目录：用于实现路由 API。
- app.js 文件：用于配置整个项目，包括路由控制。

3. Express路由机制

首先来了解下设计和实现 API 的基础：Express 的路由机制。

当浏览器访问地址 `http://127.0.0.1:3000/`时，会看到 Express 页面，此时浏览器向服务器发送了如下请求，如图 6.19 所示。

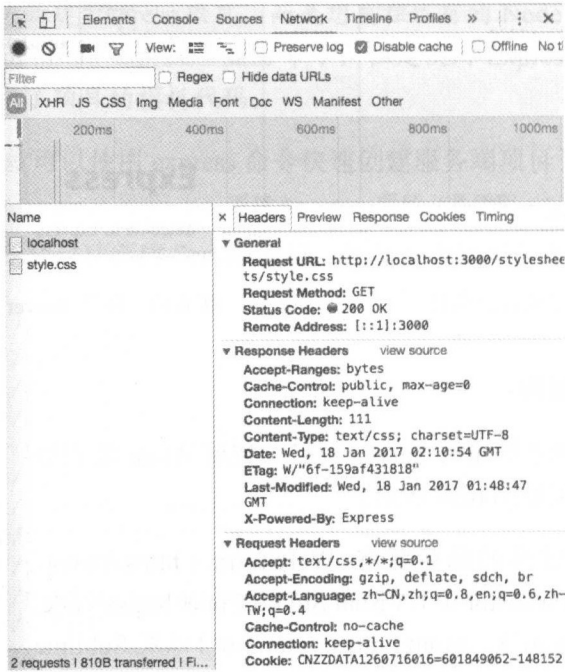



图 6.19 访问 Express 首页时的 HTTP 请求头

提示：在 Chrome 浏览器中，打开菜单“视图”→“开发者”→“JavaScript 控制台”，选择 Network 选项后单击 localhost 请求，就可以看到 HTTP 请求的详细信息。需要注意的是，不同浏览器的打开方式可能有细微差别。

服务器在接收到浏览器访问地址 `http://127.0.0.1:3000/` 的 GET 请求后, 由 `app.js` 文件中的这段代码进行处理:

```
01 // 这里省略了其他无关的代码
02
03 var index = require('./routes/index');
04 var users = require('./routes/users');
05
06 var app = express();
07
08 app.use('/', index);           // 路由 "/" 路径的请求
09 app.use('/users', users);      // 路由 users 路径的请求
10
11 // 这里省略了其他无关的代码
```

此时 “/” 路径的请求就交由 `./routes/index.js` 文件来处理, `index.js` 文件的代码如下:

```
01 var express = require('express');
02 var router = express.Router();
03
04 /* GET home page. */
05 router.get('/', function(req, res, next) {
06     res.render('index', { title: 'Express' });
07 });
08
09 module.exports = router;
```

`router` 的 `get` 方法用来处理 GET 请求, 该方法的回调函数处理和返回 HTTP 响应 `res`, `res` 的 `render()` 方法用来渲染模板文件 `index.ejs`, 模板文件 `index.ejs` 的内容如下:

```
01 <!DOCTYPE html>
02 <html>
03     <head>
04         <title><%= title %></title>
05         <link rel='stylesheet' href='/stylesheets/style.css' />
06     </head>
07     <body>
08         <h1><%= title %></h1>
09         <p>Welcome to <%= title %></p>
10     </body>
11 </html>
```

在渲染模板文件时, 会将模板中的 `<%= title %>` 替换成 `render()` 方法中传递的字典 `{ title: 'Express' }`。最终, 渲染模板文件后得到的 HTML 页面就返回给浏览器, 内容如下:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1>Express</h1>
    <p>Welcome to Express </p>
  </body>
</html>
```

6.3.2 查询商品接口

在清楚 Express 路由控制和页面渲染的过程后，就可以按照之前的设计来添加我们的商品 API 了。在正式添加之前，不知道读者是不是会好奇，想看一下此时在浏览器中访问地址 `http://127.0.0.1:3000/products` 返回的结果，效果如图 6.20 所示。

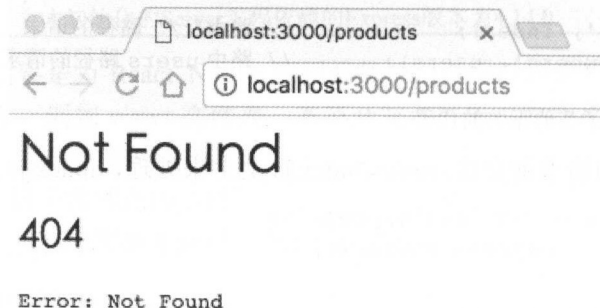


图 6.20 404 Not Found 错误

由于不存在 `/products` 的路由规则，所以服务器会在响应头中返回 404 Not Found 错误。

1. 添加接口

下面就来添加 `/products` 的路由规则。参考之前路径 `/` 的实现，首先在 `app.js` 文件中修改代码如下：

```
01 // 这里省略了没有修改的代码
02
03 var index = require('./routes/index');
04 var products = require('./routes/products');
05                                     // 引入./routes/products.js 文件
06 var users = require('./routes/users');
07
08 var app = express();
09
10 app.use('/', index);
11 app.use('/products', products);    // 添加/products 路径的请求
12 app.use('/products/:id', products); // 添加/products/(.*) 路径的请求
13
14 // 这里省略了没有修改的代码
```

然后，在 `routes` 目录下新建文件 `products.js`，并添加代码如下：

```
01 var express = require('express');
02 var router = express.Router();
03
04 /* GET products page. */
05 router.get('/', function(req, res, next) {
06     res.send('商品列表');
07 });
08
09 module.exports = router;
```

这里并没有使用响应 `res` 的 `render()` 方法，而是使用了更简便的 `send()` 方法：该方法直接返回一个字符串。此时在浏览器中访问地址 `http://127.0.0.1:3000/products`，效果如图 6.21 所示。

至此，商品 API 就已经算是正式上线了！

2. 添加数据

下面接着添加数据，并且按照本书 6.2 节 RESTful API 的设计实现各种操作的接口就可以了。

首先将图片资源复制至 `public/images` 目录下，效果如图 6.22 所示。



图 6.21 浏览器请求 `/products` 地址

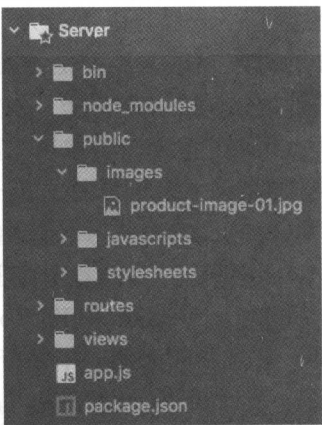


图 6.22 添加图片资源到 `public/images` 目录

然后将 React Native 应用中使用的商品数据都添加到服务器中，修改 `products.js` 文件的代码如下：

```
01 var express = require('express');
02 var router = express.Router();
03
04 var products = [
05   {
06     id: 1,
07     image: '/images/product-image-01.jpg',
08     title: '商品 1',
09     subTitle: '描述 1'
10   }, {
11     id: 2,
12     image: '/images/product-image-01.jpg',
13     title: '商品 2',
14     subTitle: '描述 2'
15   }, {
16     // 这里省略了重复的代码
17   }
18   {
19     id: 10,
20     image: '/images/product-image-01.jpg',
21     title: '商品 10',
22     subTitle: '描述 10'
23   }
24 ];
```

```

25  /* GET products page. */
26  router.get('/', function(req, res, next) {
27    res.send(JSON.stringify(products)); // 此时返回 JSON 格式的数据
28  });
29
30  module.exports = router;

```

在上述代码中,使用了 `JSON.stringify` 将 `products` 字典转换成了 JSON 格式。那么,JSON 格式又是什么呢? JSON (JavaScript Object Notation) 是一种由道格拉斯·克罗克福特 (<https://zh.wikipedia.org/wiki/%E9%81%93%E6%A0%BC%E6%8B%89%E6%96%AF%C2%B7%E5%85%8B%E7%BE%85%E5%85%8B%E7%A6%8F%E7%89%B9>) 设计的轻量级数据交换语言,以文字为基础,易于让人阅读。尽管 JSON 是 JavaScript 的一个子集,但发展至今 JSON 数据格式已与语言无关,目前很多编程语言都支持 JSON 格式数据的生成和解析。JSON 的官方 MIME 类型是 `application/json`, 文件扩展名是 `.json`。

JSON 的数据结构主要有以下几种。

- 对象 (object): 一个对象以 “{” 开始并以 “}” 结束, 一个对象包含一系列键-值对, 每个键-值对之间以 “,” 隔开。
- 键-值对 (collection): 键-值对之间使用 “:” 隔开, 形如 `{name:value}`。
- 数组 (Array): 一个或者多个值用 “,” 分区后, 使用 “[” 和 “]” 括起来就形成了数组, 形如 `[collection, collection]`。
- 字符串: 以 “” 括起来的一串字符。
- 数值: 一系列 0~9 的数字组合, 可以为负数或者小数。
- 布尔值: 可以为 `true` 或者 `false`。

综上所述, JSON 数据格式因为简单、易读以及体积小而逐渐受到 Web 开发者的青睐。

此时, 再次在浏览器中访问地址 `http://127.0.0.1:3000/products`, 效果如图 6.23 所示。

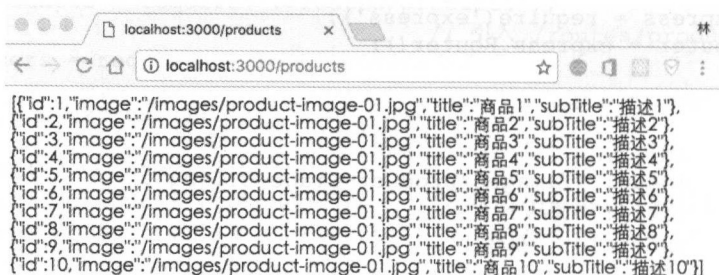


图 6.23 浏览器请求 `/products` 地址

这样, 商品的查询接口就实现了。

3. HTTP API 调试利器 Postman

下面在继续实现其他接口 (新建、更新以及删除商品) 之前, 需要提醒读者的是, 在实际 Web 开发中, 一般不使用浏览器测试 API (测试页面除外), 而是使用其他更高效的 HTTP 调试工具, 例如笔者强烈推荐的 Postman (<https://www.getpostman.com/>), 如图 6.24 所示。

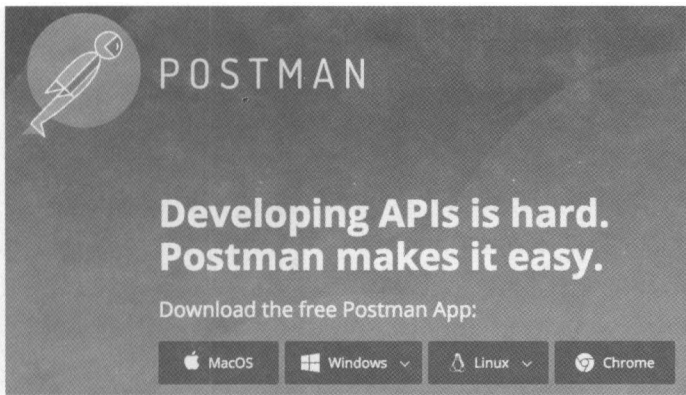


图 6.24 HTTP 调试工具 Postman

Postman 不仅支持所有的桌面系统，例如 macOS、Windows 以及 Linux，还提供了免费的 Chrome 插件版，读者可以自行在 Chrome 应用商店中搜索下载。读者根据自己的实际情况，选择相应版本安装成功后，就可以使用 Postman 来测试刚才实现的查询商品接口了。

打开 Postman，在 GET 方法的地址栏中添加商品接口的地址 `http://127.0.0.1:3000/products`，单击 Send 按钮发起 HTTP 请求，此时，HTTP 响应的结果就显示在 Body 一栏中。另外，为了更美观地查看返回的 JSON 数据，还可以将数据格式设置成 JSON，效果如图 6.25 所示。

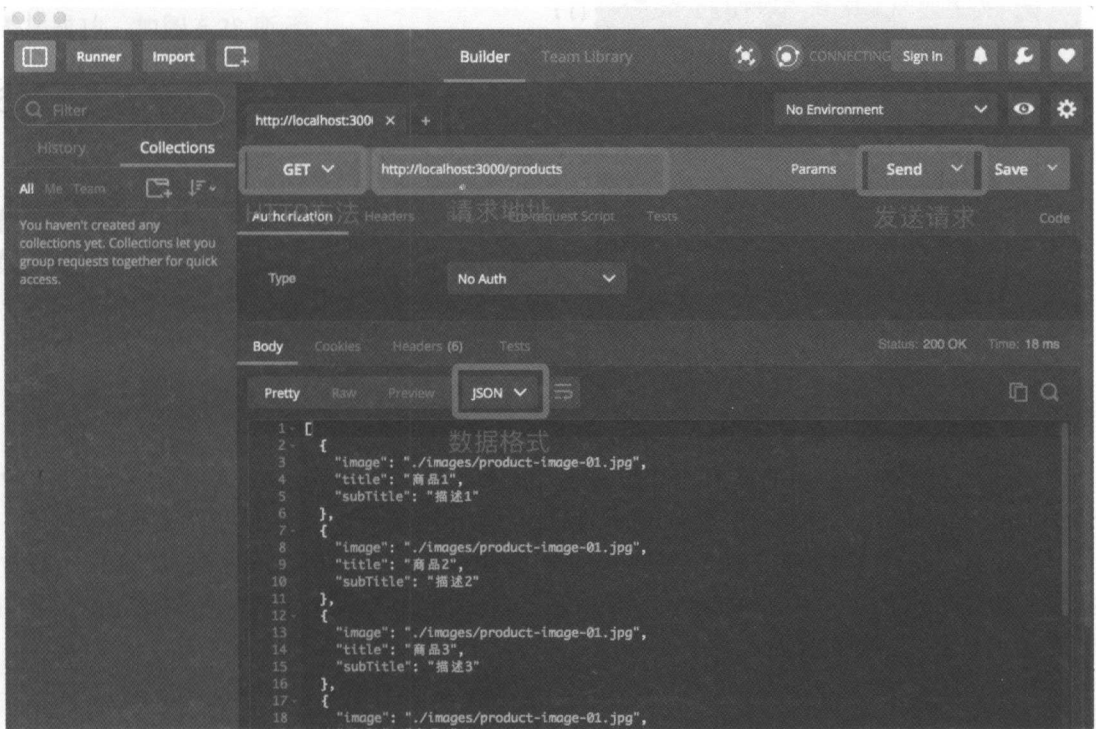


图 6.25 Postman 发送查询商品请求

测试成功后，单击 Postman 中的 Save 按钮，设置请求名称为“查询商品”，新建 Collection 名称为“商品”，效果如图 6.26 所示。

设置完毕后，单击 Save 按钮就可以保存查询商品的 HTTP 配置了。

当然，Postman 不仅仅只是让显示数据更美观而已，它的功能非常强大，包括：

- 配置 HTTP 请求参数 (Params)。
- 配置 HTTP 请求头 (Headers)。
- 按照不同格式显示返回的数据 Pretty、Raw 以及 Preview 等。
- 保存 HTTP 请求的配置便于自动化测试。
- 保存的 HTTP 请求存储在服务端，可以同步和分享。

熟悉了 Postman 之后，可以大大加快余下接口的调试进度。

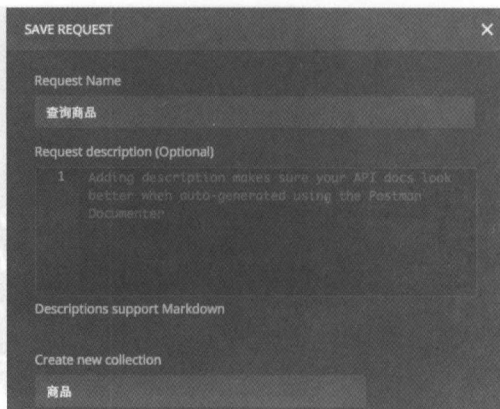


图 6.26 Postman 保存 HTTP 请求的配置

6.3.3 新建商品接口

添加新建商品的接口，根据 RESTful API 的设计规范，新建操作应该使用 HTTP 的 POST 方法。修改 products.js 代码如下：

```
01 var express = require('express');
02 var router = express.Router();
03
04 // 这里省略了没有修改的代码
05
06 // 获取商品接口
07 router.get('/', function(req, res, next) {
08     res.send(JSON.stringify(products));
09 });
10
11 // 新建商品接口
12 router.post('/', function (req, res, next) {
13     products = products.concat(req.body);
14     res.send(JSON.stringify(products));
15 });
16
17 module.exports = router;
```

新建商品的接口使用的是 HTTP 的 POST 方法，在 router.post 的回调函数中，从 HTTP 请求的 body 中取出商品数据，然后将新商品加到商品数组中，最后返回新的商品列表。

在 Postman 中模拟 HTTP 的 POST 请求。首先需要配置 HTTP 的方法为 POST，添加接口的地址 `http://127.0.0.1:3000/products`，单击请求 Body 设置选项为 raw 格式为 JSON，同时添加新商品的 JSON 数据如下：

```
[
  {
    "id": 11,
    "image": "./images/product-image-01.jpg",
    "title": "商品 11",
    "subTitle": "描述 11"
  }
]
```

Postman 的配置如图 6.27 所示。

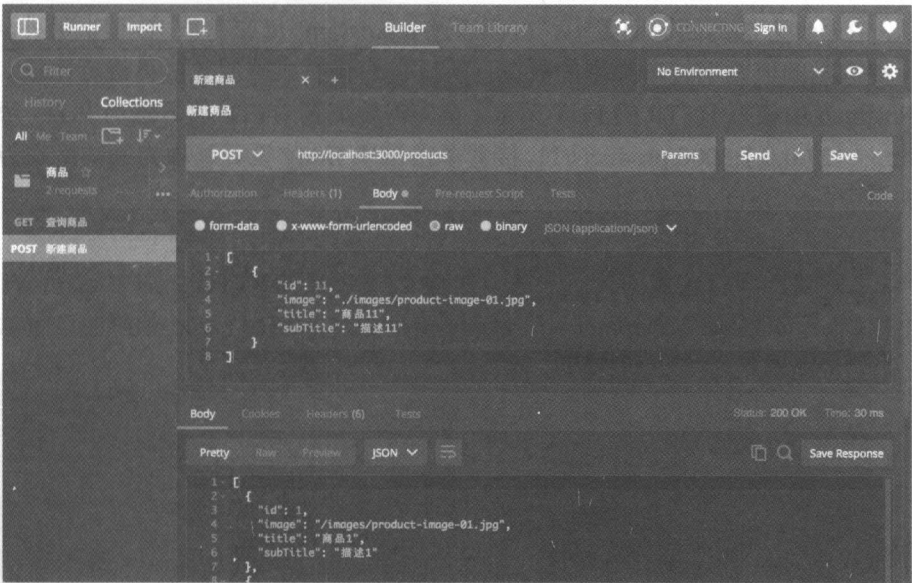


图 6.27 Postman 发送新建商品请求

使用查询商品接口验证，发现新商品已经添加成功，如图 6.28 所示。

6.3.4 更新商品接口

继续实现更新商品的接口，同理，该接口使用 HTTP 的 put 方法。修改 products.js 代码如下：

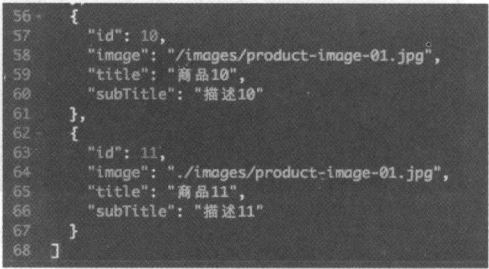


图 6.28 验证新建商品接口

```
01 var express = require('express');
02 var router = express.Router();
03
04 // 这里省略了没有修改的代码
05
06 // 获取商品接口
07
08 // 新建商品接口
09
10 // 更新商品接口
11 router.put('/:id', function (req, res, next) {
12   for (var i = 0; i < products.length; i++) {
13     if (products[i].id === parseInt(req.params.id)) {
14       products[i] = req.body;
15     }
16   }
17
18   res.send(JSON.stringify(products));
19 });
20
21 module.exports = router;
```

router.put 方法的第一个参数 “/:id” 指匹配包含该后缀的路由，而 Express 的路由机制会将当前路由的后缀，例如解析路由/products/11 得到的值 11，设置到 HTTP 请求的参数中。所以，在 put 方法的回调函数中，可以通过 req.params.id 来获取 ID 值。另外，需要注意的是，参数里的值是字符串类型的数据，所以需要转换成整型数值再进行比较。

在 Postman 中模拟 HTTP 的 PUT 请求，首先需要配置 HTTP 的方法为 PUT，添加接口的地址 http://127.0.0.1:3000/products/11，接着单击请求 Body 设置选项为 raw 格式为 JSON，同时添加新商品的 JSON 数据如下：

```
{
  "id": 11,
  "image": "../images/product-image-01.jpg",
  "title": "商品 11",
  "subTitle": "更新过的描述"
}
```

Postman 的配置如图 6.29 所示。

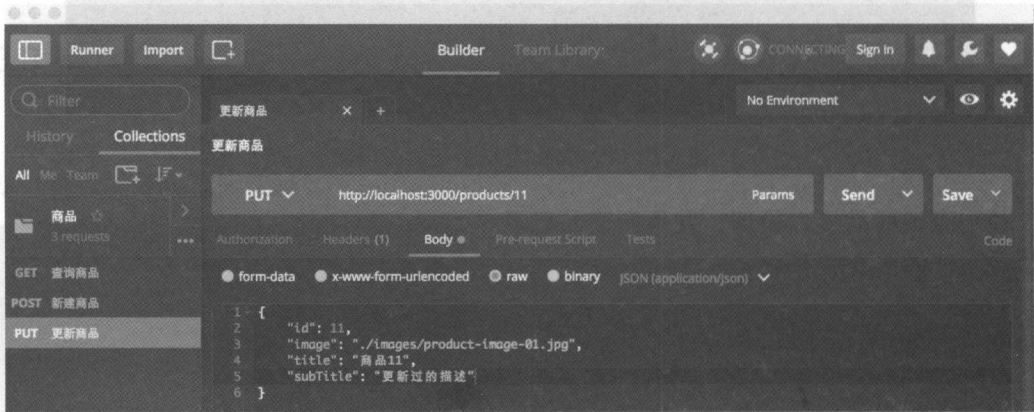


图 6.29 Postman 发送更新商品请求

使用查询商品接口验证，发现商品信息已经成功更新，如图 6.30 所示。

6.3.5 删除商品接口

使用 HTTP 的 DELETE 请求实现删除商品的接口，修改 products.js 代码如下：

```
01 var express = require('express');
02 var router = express.Router();
03
04 // 这里省略了没有修改的代码
05
06 // 获取商品接口
07
08 // 新建商品接口
09
10 // 更新商品接口
11
```

```
56 {
57   "id": 10,
58   "image": "../images/product-image-01.jpg",
59   "title": "商品10",
60   "subTitle": "描述10"
61 },
62 {
63   "id": 11,
64   "image": "../images/product-image-01.jpg",
65   "title": "商品11",
66   "subTitle": "更新过的描述"
67 }
68 ]
```

图 6.30 验证更新商品接口

```

12 // 删除商品接口
13 router.delete('/:id', function (req, res, next) {
14   for (var i = 0; i < products.length; i++) {
15     if (products[i].id === parseInt(req.params.id)) {
16       products.splice(i, 1);
17     }
18   }
19   res.send("Success");
20 });
21 };
22
23 module.exports = router;

```

在 Postman 中模拟 HTTP 的 DELETE 请求，首先需要配置 HTTP 的方法为 DELETE，然后添加接口的地址 `http://127.0.0.1:3000/products/11`。Postman 的配置如图 6.31 所示。

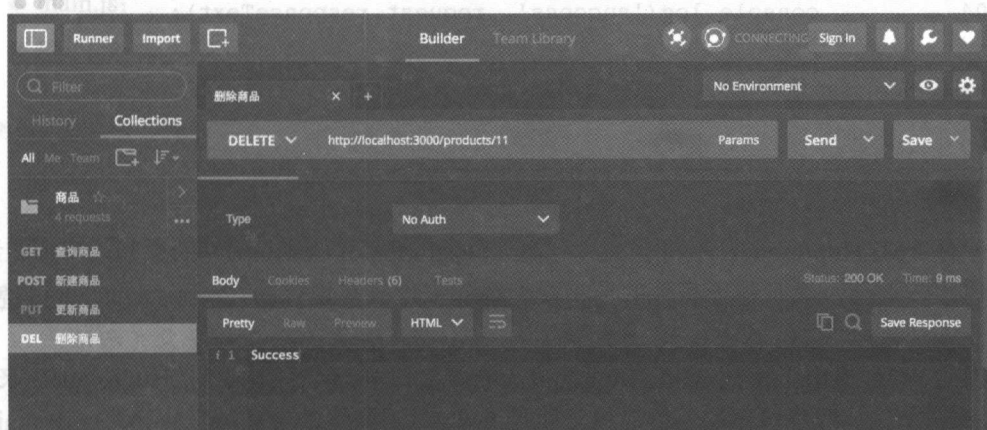


图 6.31 Postman 发送删除商品请求

使用查询商品接口验证，发现商品 11 已经被成功删除，如图 6.32 所示。

至此，获取、新建、更新以及删除商品的所有接口都已经实现并测试通过，使用 Node 进行服务器开发的工作也将告一段落。接下来会继续回到移动端开发中，来完善我们的 React Native 应用。

```

50 {
51   "id": 9,
52   "image": "/images/product-image-01.jpg",
53   "title": "商品9",
54   "subTitle": "描述9"
55 },
56 {
57   "id": 10,
58   "image": "/images/product-image-01.jpg",
59   "title": "商品10",
60   "subTitle": "描述10"
61 }
62 ]

```

图 6.32 验证删除商品接口


提示：如果读者想更深一步了解 Node 开发，可以参考 Express 官方文档 (<http://www.expressjs.com.cn/>) 或者 GitHub 上的其他资源，例如 awesome-nodejs (<https://github.com/sindresorhus/awesome-nodejs>)。

6.4 网络前后端交互的原理 fetch

在 6.3 节中使用 Node 完成了服务器接口的开发之后，现在就可以在 React Native 应用中添加与服务器数据交互的功能了。React Native 为了实现网络交互的功能，提供了如下两种 API。

- XMLHttpRequest;
- fetch。


其中, XMLHttpRequest 一直是 Web 开发者的亲密助手, 当谈及 AJAX 技术的时候, 通常意思就是基于 XMLHttpRequest 的 AJAX。XMLHttpRequest 的用法如下。

 **小知识:** AJAX (Asynchronous JavaScript And XML 即异步 JavaScript 和 XML) 不是一门新的开发语言, 而是一种基于已有技术和标准的新方法。其最大的优点是可以在不重新加载整个页面的情况下, 与服务器交换数据并更新部分网页内容。

```
01 var request = new XMLHttpRequest();
02 request.onreadystatechange = (err) => {
03   if (request.status === 200) {
04     console.log('success', request.responseText);
05   } else {
06     console.warn('fail');
07   }
08 };
09
10 request.open('GET', 'http://127.0.0.1:3000/');
11 request.send();
```

但是 XMLHttpRequest 不符合关注分离 (Separation of Concerns) 的原则, 配置和调用方式也比较混乱, 而且基于事件的异步模型写起来也没有现代的 Promise、generator/yield 以及 async/await 友好。

因此 W3C 提出了替代 XMLHttpRequest 的新接口 fetch。相比 XMLHttpRequest 来说, fetch 是一个封装程度更高的网络 API, 而且 fetch 对于网络的异步处理完全基于 Promise, API 更加简洁友好。

 **小知识:** Promise 是一套异步操作的处理机制, 可以消除“回调金字塔”, 即 Callback Hell (<http://callbackhell.com/>) 问题, 并且更容易控制异步操作的执行顺序。

使用 fetch 接口实现请求服务器首页的代码如下:

```
01 const req = new Request('http://127.0.0.1:3000/', {method: 'GET'});
02 fetch(req).then((res) => {
03   return res;
04 }).then((result, done) => {
05   console.log("result = " + JSON.stringify(result));
06   if (!done) {
07     console.log("sucess");
08   } else {
09     console.warn('fail');
10   }
11 };
```

6.5 App 从服务器获取数据

在了解完 AJAX/fetch 的概念和用法之后, 就可以使用 fetch 接口为 React Native 应用添加网络功能了。

(1) 先创建 React Native 项目并安装依赖包。


```
react-native init ch05 // 新建 React Native 项目 ch05
cd ch05
npm install
```

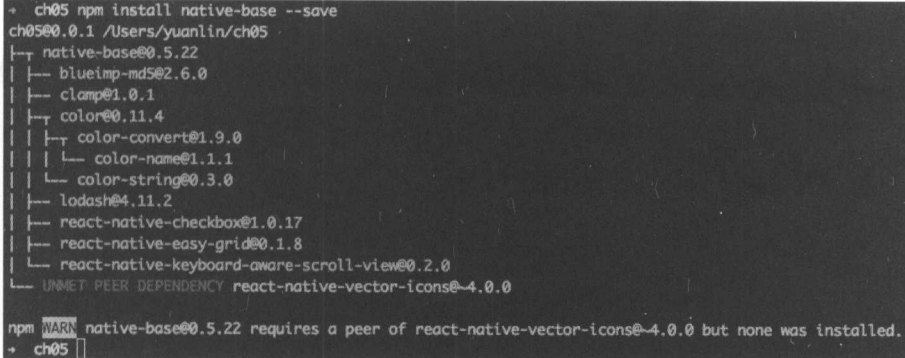
(2) 将 ch04 项目中如下目录或文件都复制到 ch05 文件夹中。

- app.js;
- detail.js;
- home.js;
- images;
- index.android.js;
- index.ios.js;
- main.js;
- more.js。

(3) 安装第三方依赖包 NativeBase、react-native-vector-icons 以及 react-native-swiper, 命令如下:

```
npm install native-base --save
react-native link react-native-vector-icons
npm install react-native-swiper --save
```

 提示: 如果安装 NativeBase 时发生找不到依赖 react-native-vector-icons 的错误(如图 6.33 所示),需要安装 react-native-vector-icons 依赖包,使用命令 `npm install react-native-vector-icons --save` 即可。



```
+ ch05 npm install native-base --save
ch05@0.0.1 /Users/yuanlin/ch05
├─┬ native-base@0.5.22
│   ├── blueimp-md5@2.6.0
│   ├── clamp@1.0.1
│   ├── color@0.11.4
│   │   ├── color-convert@1.9.0
│   │   │   ├── color-name@1.1.1
│   │   │   └── color-string@0.3.0
│   │   └── lodash@4.11.2
│   ├── react-native-checkbox@1.0.17
│   ├── react-native-easy-grid@0.1.8
│   ├── react-native-keyboard-aware-scroll-view@0.2.0
│   └── UNMET PEER DEPENDENCY react-native-vector-icons@4.0.0
npm WARN native-base@0.5.22 requires a peer of react-native-vector-icons@4.0.0 but none was installed.
+ ch05
```

图 6.33 安装 NativeBase 错误

(4) 修改 index.ios.js 和 index.android.js 中的代码如下:

```
01 import React, {Component} from 'react';
02 import {AppRegistry} from 'react-native';
03 import app from './app';
04
05 AppRegistry.registerComponent('ch05', () => app); // 修改第一个参数为'ch05'
```

此时,使用命令 `react-native run-ios` 或者 `react-native run-android` 运行 ch05 项目,确保 ch04 项目的实现成功移植到 ch05 项目中,效果如图 6.34 所示。



图 6.34 ch05 项目运行效果

6.5.1 获取商品信息

准备工作完毕后，就可以修改代码添加功能了。

1. 获取网络数据

首先，在 `home.js` 文件中添加获取网络数据的代码如下：

```
01 // 这里省略了没有修改的代码
02
03 const SERVER_URL = 'http://localhost:3000/';
04 const PRODUCT_API = 'products/';
05
06 export default class home extends Component {
07   // 这里省略了没有修改的代码
08
09   componentDidMount() {
10     this._fetchProducts();
11   }
12
13   // 这里省略了没有修改的代码
14
15   _fetchProducts = () => {
16     const req = new Request(SERVER_URL + PRODUCT_API, {method:
17       'GET'});
18     console.log('request: ', SERVER_URL + PRODUCT_API);
19     fetch(req).then((res) => {
20       return res.json(); // 将返回的数据转换成 JSON 格式
21     }).then((result, done) => {
22       if (!done) {
23         console.log('result: ' + JSON.stringify(result));
24       }
25     })
26   }
```

```

24     });
25   }
26 }

```

🔔注意：应用要想从服务器成功获取数据，首先必须要启动服务器程序，可以打开一个新的终端，进入前面开发的 Server 项目中，使用 `npm start` 命令启动服务。

重新加载应用，然后打开调试选项 `Debug JS Remotely`，此时可以在 Chrome 浏览器的调试终端看到打印信息如图 6.35 所示。

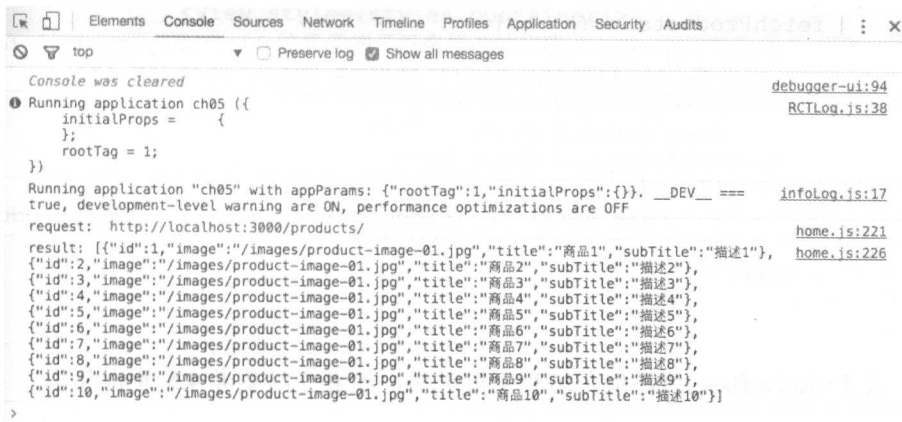


图 6.35 获取商品列表信息

至此，React Native 应用和服务器的连接都已经成功建立起来了！

2. 展示数据

修改应用的逻辑，将从服务器获取的数据展示到页面上。修改 `home.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04   constructor(props) {
05     super(props);
06     this.state = {
07       // 这里省略了没有修改的代码
08       products: [] // 删除 products 内容
09     };
10   }
11
12   // 这里省略了没有修改的代码
13
14   _renderRow = (product) => {
15     return (
16       <ListItem button onPress={() => {
17         const {navigator} = this.props;
18         if (navigator) {
19           navigator.push({
20             name: 'detail',
21             component: Detail,
22             params: {
23               productTitle: product.title
24             }
25           });
26         }
27       }}

```

```

27         }
28     }>
29     <Thumbnail square size={40} source={{
30         uri: SERVER_URL + product.image // 设置Thumbnail的uri
31     }}/>
32     <Text>{product.title}</Text>
33     <Text note>{product.subTitle}</Text>
34 </ListItem>
35 );
36 }
37
38 _fetchProducts = () => {
39     const req = new Request(SERVER_URL + PRODUCT_API, {method: 'GET'});
40     console.log('request: ', SERVER_URL + PRODUCT_API);
41     fetch(req).then((res) => {
42         return res.json();
43     }).then((result, done) => {
44         if (!done) {
45             this.setState({products: result}); // 更新 products
46         }
47     });
48 }
49 }

```

 **提示：**关于 NativeBase 的详细介绍和使用说明，读者可以参考 4.5 节的内容。

重新加载应用，待应用获取网络数据和图片地址后，React Native 应用显示的就是从服务端的数据了，效果如图 6.36 所示。

这里如果不使用 NativeBase 的 List 组件，而是使用 React Native 原生的 ListView 组件的话，可能会遇到以下的警告：

Warning: In next release empty section headers will be rendered. In this release you can use 'enableEmptySections' flag to render empty section headers

效果如图 6.37 所示。



图 6.36 应用成功获取服务器数据和图片

图 6.37 使用 ListView 组件时的 enableEmptySections 警告

为了解决这个警告，可以按照警告提示的方法，在 `ListView` 组件中配置属性 `enableEmptySections`。示例代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04   // 这里省略了没有修改的代码
05
06   render() {
07     return (
08       <View style={styles.container}>
09         // 这里省略了没有修改的代码
10         <View style={styles.products}>
11           <ListView dataSource={this.state.dataSource}
12             onRefresh={this._onRefresh}
13             renderRow={this._renderRow}
14             renderSeparator={this._renderSeparator}
15             refreshControl={this._renderRefreshControl()}
16             enableEmptySections={true}
17           // 配置 enableEmpty Sections
18         </View>
19       </View>
20     );
21   }
22
23   // 这里省略了没有修改的代码
24 }

```

6.5.2 更新商品信息

为应用成功添加了第一个获取商品的网络接口之后，接着添加更新商品的功能。

1. 完善商品详情页面

需要完善商品详情页面，用来显示和编辑商品信息。修改 `detail.js` 代码如下：

```

01 import React, {Component} from 'react';
02 import {StyleSheet, View, Text, TouchableOpacity} from 'react-native';
03
04 export default class detail extends Component {
05   render() {
06     return (
07       <View style={styles.container}>
08         <TouchableOpacity onPress={this._pressBackButton.bind(
09           this)}>
10           <Text style={styles.back}>返回</Text>
11         </TouchableOpacity>
12         <Text style={styles.text}>id: {this.props.product.id}</Text>
13         <Text style={styles.text}>title: {this.props.product.
14           subTitle}</Text>
15         <Text style={styles.text}>subTitle: {this.props.product.
16           subTitle}</Text>
17         <Text style={styles.text}>image: {this.props.product.
18           image}</Text>
19       </View>
20     );
21   }
22 }

```

```

17     }
18   }
19
20   const styles = StyleSheet.create({
21     container: {
22       flex: 1,
23       justifyContent: 'center'
24     },
25     text: {
26       fontSize: 30
27     },
28     back: {
29       fontSize: 20,
30       color: 'blue'
31     }
32   });

```

同时，还需要修改传入商品详情页面的参数。修改 `home.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04   // 这里省略了没有修改的代码
05
06   _renderRow = (product) => {
07     return (
08       <ListItem button onPress={() => {
09         const {navigator} = this.props;
10         if (navigator) {
11           navigator.push({
12             name: 'detail',
13             component: Detail,
14             params: {
15               product: product
16             }
17           });
18         }
19       } >
20       // 这里省略了没有修改的代码
21     </ListItem>
22   );
23 }
24
25 // 这里省略了没有修改的代码
26 }

```

重新加载应用，然后单击某一商品，即可跳转到商品详情页面，效果如图 6.38 所示。

2. 编辑商品

实现了基本的原型和流程之后，就可以完善商品详情页面，添加编辑功能了。修改 `detail.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class detail extends Component {
04   constructor(props) {
05     super(props);
06     this.state = {

```



图 6.38 商品详情页面原型

```

07         productID: '' + this.props.product.id,
08         productTitle: this.props.product.title,
09         productSubTitle: this.props.product.subTitle
10     }
11 }
12
13 render() {
14     return (
15         <View style={styles.container}>
16             // 这里省略了没有修改的代码
17             <View style={styles.line}>
18                 <Text style={styles.text}>ID:</Text>
19                 <TextInput style={styles.input}
20                     value={this.state.productID} onChangeText={(text)
21                     => {
22                         this.setState({productID: text});
23                     }}></TextInput>
24             </View>
25             <View style={styles.line}>
26                 <Text style={styles.text}>title:</Text>
27                 <TextInput style={styles.input}
28                     value={this.state.productTitle} onChangeText=
29                     {(text) => {
30                         this.setState({productTitle: text});
31                     }}></TextInput>
32             </View>
33             <View style={styles.line}>
34                 <Text style={styles.text}>subTitle:</Text>
35                 <TextInput style={styles.input}
36                     value={this.state.productSubTitle} onChangeText=
37                     {(text) => {
38                         this.setState({productSubTitle: text});
39                     }}></TextInput>
40             </View>
41             <View style={styles.line}>
42                 <Text style={{fontSize: 20}}>
43                     image: {this.props.product.image}</Text>
44             </View>
45         </View>
46     );
47 }
48 }

```

同时，修改 detail.js 文件的样式和布局代码如下：

```

01 const styles = StyleSheet.create({
02     container: {
03         flex: 1,
04         marginTop: 100
05     },
06     line: {
07         flexDirection: 'row'
08     },
09     text: {
10         width: 100,
11         fontSize: 20
12     },
13     input: {
14         flex: 1,
15         borderColor: 'gray',
16         borderWidth: 2
17     },
18 });

```

```

19      // 这里省略了没有修改的代码
20    });

```

重新加载应用，然后单击某一商品，此时商品详情页面的效果如图 6.39 所示。

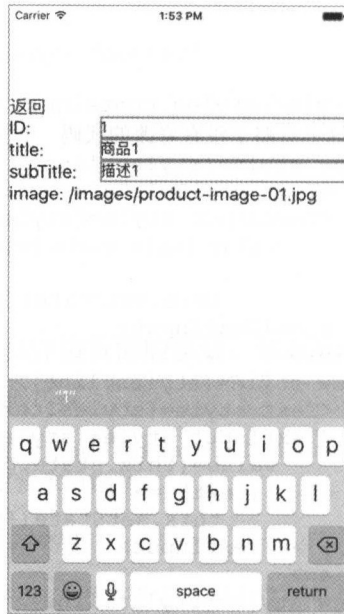


图 6.39 商品详情页面

3. 保存编辑

在添加编辑功能之后，接着再添加一个保存结果的“保存”按钮。修改 detail.js 代码如下：

```

01  // 这里省略了没有修改的代码
02
03  const SERVER_URL = 'http://localhost:3000/';
04  const PRODUCT_API = 'products/';
05
06  export default class detail extends React.Component {
07    // 这里省略了没有修改的代码
08
09    render() {
10      return (
11        <View style={styles.container}>
12          // 这里省略了没有修改的代码
13          <Button title='保存' onPress={this._updateProduct}>
14            </Button>
15        </View>
16      );
17    }
18
19    _updateProduct = () => {
20      const req = new Request(SERVER_URL + PRODUCT_API + this.state.
21        productID, {
22        method: 'PUT',
23        headers: { // 设置 HTTP 请求头的数据格式为 JSON
24          'Accept': 'application/json',

```



```

23         'Content-Type': 'application/json'
24     },
25     body: JSON.stringify({
26         'id': parseInt(this.state.productID),
27         'title': this.state.productTitle,
28         'subTitle': this.state.productSubTitle,
29         'image': this.props.product.image
30     })
31 });
32 fetch(req).then((res) => {
33     return res.json();
34 }).then((result, done) => {
35     if (!done) {
36         Alert.alert('保存成功', null, null);
37     } else {
38         Alert.alert('保存失败', null, null);
39     }
40 });
41 }
42 }

```

编辑商品的描述，修改“描述1”为 New Description，然后单击“保存”按钮，应用会发送更新商品信息的 HTTP 请求，请求成功后效果如图 6.40 所示。

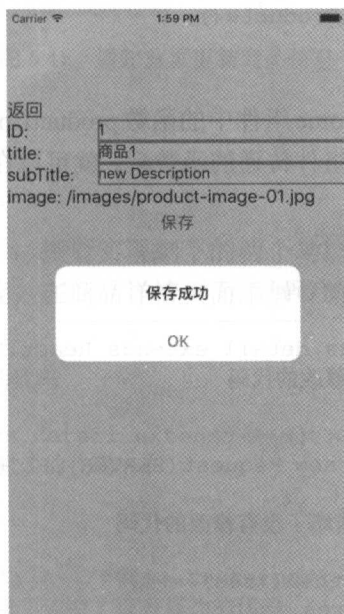


图 6.40 更新商品信息

4. 通知首页重新获取商品列表

虽然更新商品信息成功了，但是首页的数据并没有更新，因此在保存成功的同时还需要通知首页重新获取商品列表。修改 home.js 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04     // 这里省略了没有修改的代码

```

```

05
06   _renderRow = (product) => {
07     return (
08       <ListItem button onPress={() => {
09         const {navigator} = this.props;
10         if (navigator) {
11           navigator.push({
12             name: 'detail',
13             component: Detail,
14             params: {
15               product: product,
16               productUpdated: this._productUpdated
17             }
18           });
19         }
20       }>
21       // 这里省略了没有修改的代码
22     </ListItem>
23   );
24 }
25 // 这里省略了没有修改的代码
26
27 _productUpdated = () => {
28   this._fetchProducts();
29 }
30 }

```

在上述 `home.js` 代码中,将 `home` 组件中的函数 `productUpdated()` 作为参数传递给了 `detail` 组件, `detail` 组件接收到上一个组件传递的函数后,就可以在恰当的时机调用该函数,这样就实现了组件间的通信。

同时,修改 `detail.js` 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 export default class detail extends React.Component {
04   // 这里省略了没有修改的代码
05
06   _updateProduct = () => {
07     const req = new Request(SERVER_URL + PRODUCT_API + this.state.
08       productID, {
09       // 这里省略了没有修改的代码
10     });
11     fetch(req).then((res) => {
12       return res.json();
13     }).then((result, done) => {
14       if (!done) {
15         this.props.productUpdated(); // 通知 home 组件商品信息
16                                     更新成功
17       } else {
18         Alert.alert('保存成功', null, null);
19       }
20     });
21   }
22 }

```

此时,在商品页面更新商品信息成功后,返回到首页,发现此时首页的商品列表也已

经更新了，效果如图 6.41 所示。



图 6.41 通知首页更新商品信息

6.5.3 新建商品

完成了获取和更新操作之后，接着实现剩下的两个功能：新建和删除商品。

首先，实现新建商品的功能。在商品详情页面添加“新建”按钮，修改 detail.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class detail extends React.Component {
04   // 这里省略了没有修改的代码
05
06   render() {
07     return (
08       <View style={styles.container}>
09         // 这里省略了没有修改的代码
10         <Button title='新建' onPress={this._createProduct}>
11           </Button>
12       </View>
13     );
14   }
15
16   _createProduct = () => {
17     const req = new Request(SERVER_URL + PRODUCT_API, {
18       method: 'POST',
19       headers: { // 设置 HTTP 请求头的数据格式为 JSON
20         'Accept': 'application/json',
21         'Content-Type': 'application/json'
22       },
23       body: JSON.stringify({
```

```

23         'id': parseInt(this.state.productID),
24         'title': this.state.productTitle,
25         'subTitle': this.state.productSubTitle,
26         'image': this.props.product.image
27     })
28   });
29   fetch(req).then((res) => {
30     return res.json();
31   }).then((result, done) => {
32     if (!done) {
33       this.props.productUpdated();
34       Alert.alert('新建成功', null, null);
35     } else {
36       Alert.alert('新建失败', null, null);
37     }
38   });
39 }
40 }

```

然后单击某一商品进入商品页面，修改商品 ID 为“11”，商品标题为“商品 11”，商品描述为“描述 11”，接着，单击“新建”按钮就新建了一个商品，效果如图 6.42 所示。单击 OK 按钮关闭提示框后，返回到首页，滑动至商品列表底部，就可以看到新添加的商品 11 了，效果如图 6.43 所示。

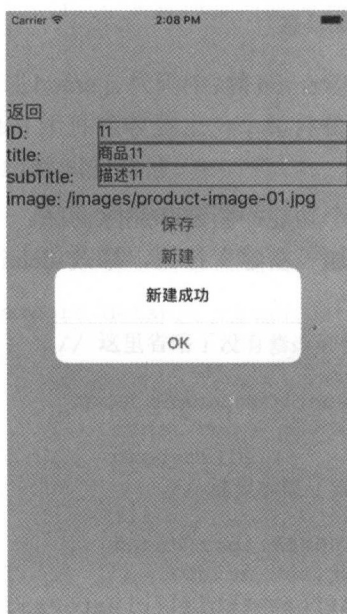


图 6.42 新建商品



图 6.43 在首页查看新建商品

6.5.4 删除商品

同样，为了添加删除商品功能，首先要在商品详情页面添加“删除”按钮，修改 detail.js 代码如下：

```

01 // 这里省略了没有修改的代码
02

```

```

03 export default class detail extends React.Component {
04   // 这里省略了没有修改的代码
05
06   render() {
07     return (
08       <View style={styles.container}>
09         // 这里省略了没有修改的代码
10         <Button title='删除' onPress={this._deleteProduct}>
11           </Button>
12       </View>
13     );
14   }
15
16   _deleteProduct = () => {
17     const req = new Request(SERVER_URL + PRODUCT_API + this.state.
18       productID, {method: 'DELETE'});
19     fetch(req).then((res) => {
20       return res; // 此时返回的数据不是 JSON 格式，所以不做处理
21     }).then((result, done) => {
22       if (!done) {
23         this.props.productUpdated();
24         Alert.alert('删除成功', null, null);
25       } else {
26         Alert.alert('删除失败', null, null);
27       }
28     });
29   }
30 }
31
32
33
34
35
36
37
38
39
40 }

```

单击首页商品列表的商品 11，进入商品页面，然后单击“删除”按钮，此时发送删除商品的请求，请求成功后效果如图 6.44 所示。单击 OK 按钮关闭提示框后，返回到首页，滑动至商品列表底部，就可以看到商品 11 被删除了，效果如图 6.45 所示。

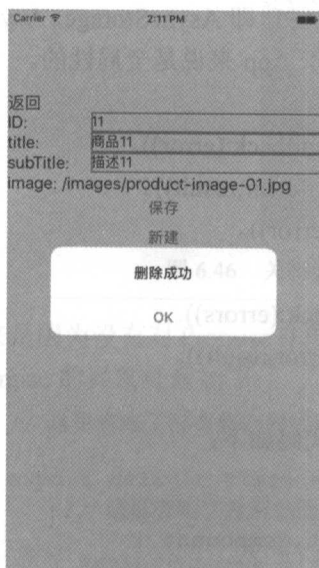


图 6.44 删除商品



图 6.45 在首页查看删除商品

至此，和服务器交互的所有网络接口都已经实现了。

从上述使用 Node 开发服务器和使用 React Native 开发应用的过程中，想必读者已经感

受到了 JavaScript 语言的强大，不仅仅实现了移动端的跨平台开发，还解决了前后端“跨端”的开发问题。当然，这不仅要归功于 Facebook、Google 类开放创新的公司，还应该感谢为这些开源项目做出贡献的所有开发者，有了他们的支持，才有了 Node 和 React Native 等平台的快速发展和普及，在此，笔者再一次对他们表示敬意。

6.6 App 数据的本地化存储

我们的应用已经具有了从网络获取数据的能力，但是如果网络断开怎么办呢？为了更好的用户体验，通常在未连接网络的设备上打开应用会显示上一次获取到的数据，因此当获取数据成功后，需要存储至本地。下面就来给应用添加数据本地化存储的能力。

React Native 开发中常见的本地数据存储的方法如下。

- **AsyncStorage**: React Native 提供的键-值存储系统，接口和功能简单，只能存储字符串数据。
- **SQLite** (<https://www.sqlite.org/>): 原生应用开发中比较常见的数据存储方法。
- **Realm** (<https://realm.io/>): 一种新兴的数据存储方法，使用简单但功能却很强大。
- 其他: 例如自定义 API 使用原生的存储方法、使用 **Redux** (<https://github.com/reactjs/redux>) 生态圈的本地持久化实现 **redux-persist** (<https://github.com/rt2zz/redux-persist>)。

6.6.1 AsyncStorage 异步键值存储

为了实现数据储存，可以使用 React Native 提供的 API，即 **AsyncStorage**。**AsyncStorage** 是一个简单的、异步的、持久化的键-值存储系统，对于 App 来说是全局性的。

AsyncStorage 的常用接口有以下几种。

- 设置键-值对: `setItem(key : string, value : string, callback:(error))`。
- 读取键所对应的值: `getItem(key : string, callback:(error,result))`。
- 移除键-值对: `removeItem(key : string, callback:(error))`。
- 清除所有内容: `clear(callback:(error))`。
- 设置多项键-值对: `multiSet(keyValuePairs, callback:(errors))`。
- 读取多个键对应的值: `multiGet(keys, callback:(error,result))`。

这里主要使用 `setItem` 和 `getItem` 两个接口。

首先，使用 `setItem` 来存储数据，修改 `home.js` 的代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class home extends React.Component {
04   // 这里省略了没有修改的代码
05
06   _fetchProducts = () => {
07     const req = new Request(SERVER_URL + PRODUCT_API, {method:
08       'GET'});
09     fetch(req).then((res) => {
10       return res.json();
11     });
12   }
13 }
```

```

10      }).then((result, done) => {
11          if (!done) {
12              AsyncStorage.setItem('products', JSON.stringify(result)).
then((err) => {
13                  if (err) {
14                      Alert.alert('err: ' + err, null, null);
15                  }
16                  this.setState({products: result});
17              });
18          }
19      });
20  }
21  }

```

然后关闭设备网络或服务器，再重新发送获取商品请求会发生如下警告：

Possible Unhandled Promise Rejection (id: 0): Network request failed

效果如图 6.46 所示。

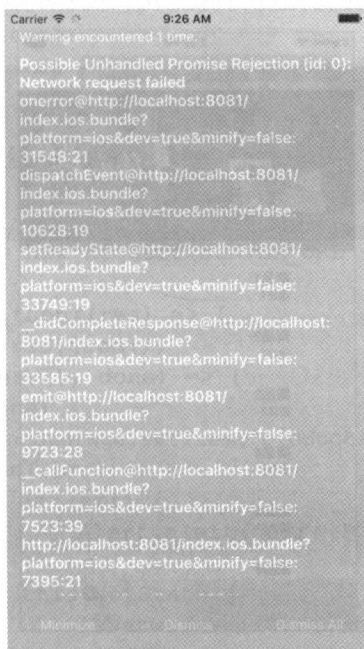


图 6.46 关闭设备网络或服务器后发起获取商品请求

原来是因为没有对 Promise 在网络请求失败时的异常做捕获和处理。修改 home.js 代码，添加 Promise 的异常处理如下：

```

01  // 这里省略了没有修改的代码
02
03  export default class home extends React.Component {
04      // 这里省略了没有修改的代码
05
06      _fetchProducts = () => {
07          const req = new Request(SERVER_URL + PRODUCT_API, {method:
'GET'});
08          fetch(req).then((res) => {
09              return res.json();
10          }).then((result, done) => {
11              if (!done) {

```



```

12         AsyncStorage.setItem('products', JSON.stringify(result)).
13         then((err) => {
14             if (err) {
15                 Alert.alert('err: ' + err, null, null);
16             }
17             this.setState({products: result});
18         });
19     }).catch((err) => { // Promise 异常处理
20         AsyncStorage.getItem('products').then((values) => {
21             this.setState({
22                 products: JSON.parse(values)
23             });
24         });
25     });
26 }
27 }

```

在断开设备或服务器的情况下重新加载应用，效果如图 6.47 所示。



图 6.47 在 Promise 异常处理中读取本地存储数据

不过这里还有一个可以改进的地方，由于 `Image` 组件的 `uri` 设置的是服务器地址，所以当网络连接不上时，虽然数据可以从本地获取并正常显示，但是图片却显示不出来。为了更好的体验，可以在获取不到网络图片时加载本地的默认图片。修改 `home.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class home extends React.Component {
04     constructor(props) {
05         super(props);
06         this.state = {
07             isNetworkValid: false, // 添加网络状态的标识
08             // 这里省略了没有修改的代码
09         };
10     }
11 }

```

```

12 // 这里省略了没有修改的代码
13
14 _renderRow = (product) => {
15   // 网络正常时加载网络图片，网络断开时加载本地图片
16   const imageComponent = this.state.isNetworkValid
17     ? <Thumbnail square size={40} source={{
18       uri: SERVER_URL + rowData.image}}/>
19     : <Thumbnail square size={40}
20       source={require('./images/product-image-01.jpg')}/>;
21
22
23   return (
24     <ListItem button onPress={() => {
25       // 这里省略了没有修改的代码
26     }}>
27     {ImageComponent} // 使用上面创建的 ImageComponent
28     <Text>{product.title}</Text>
29     <Text note>{product.subTitle}</Text>
30   </ListItem>
31   );
32 }
33
34 // 这里省略了没有修改的代码
35
36 _fetchProducts = () => {
37   const req = new Request(SERVER_URL + PRODUCT_API, {method:
38     'GET'});
39   fetch(req).then((res) => {
40     return res.json();
41   }).then((result, done) => {
42     if (!done) {
43       AsyncStorage.setItem('products', JSON.stringify(result)).
44       then((err) => {
45         if (err) {
46           Alert.alert('err: ' + err, null, null);
47         }
48         this.setState({
49           isNetworkValid: true, // 设置网络标识为 true
50           products: result
51         });
52       });
53     }
54   }).catch((err) => {
55     AsyncStorage.getItem('products').then((values) => {
56       this.setState({
57         isNetworkValid: false, // 设置网络标识为 false
58         products: JSON.parse(values)
59       });
60     });
61   });
62 }

```

此时，应用在网络连接时从服务器获取数据和图片资源，网络断开时从本地读取数据和加载图片，效果如图 6.48 所示。



图 6.48 根据网络状态加载不同位置的数据和图片

6.6.2 SQLite 数据库

AsyncStorage 比较适合存放较小的简单数据，如果想要存放数据结构较复杂的数据，可以使用移动平台常用的数据库 SQLite。

SQLite (<https://www.sqlite.org/>) 是一个开源的嵌入式关系数据库，实现自包容、零配置、支持事务的 SQL 数据库引擎。其特点是高度便携、使用方便、结构紧凑、高效以及可靠。SQLite 是原生应用开发中常用的数据存储方法，在 React Native 开发中，可以使用第三方库来支持 SQLite。本书将以 react-native-sqlite-storage (<https://github.com/andpor/react-native-sqlite-storage>) 为例，介绍 SQLite 在 React Native 开发中的使用。

提示：除了本书介绍的 react-native-sqlite-storage，读者还可以在 JS.COACH (<https://js.coach/react-native>) 上搜索 SQLite 关键字，查找更多的 SQLite 第三方库。

首先，添加 react-native-sqlite-storage 到 React Native 项目中，命令如下：

```
npm install --save react-native-sqlite-storage // 安装第三方库
```

注意：截止笔者完稿时，react-native-sqlite-storage 最新版本 3.1.3 还不支持最新的 React Native 版本 0.40，所以下面主要演示 react-native-sqlite-storage 的用法，后续版本更新后，读者可自行尝试实际运行效果。

要使用 SQLite 数据库，必须实现打开和关闭数据库的操作，代码如下：

```
01 // 打开数据库
02 db = SQLiteStorage.openDatabase('product.db', '1.0', 'app-db', -1, ()
=> {
03   console.log("DB Opened");
```

```

04 }, (err) => {
05   console.log("SQL Error: " + err);
06 });
07 if (db) {
08   db.transaction((tx) => {
09     // 创建数据库表 product
10     tx.executeSql('CREATE TABLE IF NOT EXISTS '
11       + 'product'
12       + '('
13       + 'id INTEGER PRIMARY KEY NOT NULL,'
14       + 'title VARCHAR,'
15       + 'subTitle VARCHAR,'
16       + 'image VARCHAR'
17       + ');', [],
18     () => {
19       console.log("SQL executed fine");
20     }, (err) => {
21       console.log("SQL Error: " + err);
22     });
23   });
24 }
25 }
26
27 // 关闭数据库
28 if (db) {
29   console.log("DB Closed");
30   db.close();
31 }
32 db = null;

```

在成功打开数据库后, 就可以使用 INSERT 和 SELECT 语句进行数据的添加和查询操作了, 代码实现如下:

```

01 // 插入数据
02 if (db) {
03   db.executeSql(
04     'INSERT INTO ' + product + ' (title, subTitle, image) VALUES'
05     '(?, ?, ?)',
06     ['商品 1', '描述 1', 'images/advertisement-image-01.jpg'],
07     () => {
08       console.log("SQL executed fine");
09     }, (err) => {
10       console.log("SQL Error: " + err);
11     });
12 }
13
14 // 查询数据
15 if (db) {
16   db.executeSql(
17     'SELECT * FROM ' + 'product',
18     [],
19     (results) => {
20       console.log("SQL executed fine " + results);
21     }, (err) => {
22       console.log("SQL Error: " + err);
23     });
24 }

```


在使用 react-native-sqlite-storage 的过程中可以发现, React Native 开发中使用的 SQLite 其实都是基于原生平台 SQLite 的封装, 稳定性和开发效率并不是很高, 所以尽管原生应用

开发中常常使用 SQLite，但是在实际的 React Native 开发中，本地的数据存储并不推荐使用 SQLite 的方式。

那么有什么更好的处理复杂数据存储的办法呢？答案就是 6.6.3 节要介绍的主角 Realm。

6.6.3 Realm 数据库

Realm (<https://realm.io/>) 是由 Y Combinator 公司孵化出来的一款跨平台移动数据库，为了彻底解决性能问题，核心数据引擎使用 C++ 打造。Realm 相比 SQLite 而言更快、更简洁、使用也更加简单。


 **小知识：** Y Combinator 成立于 2005 年，是美国著名创业孵化器，Y Combinator 扶持初创企业并为其提供创业指南。国内成立于 2009 年的创新工场，就参考了 Y Combinator 的模式。

Realm 现在已经支持几乎所有的移动开发平台，例如 iOS、Android、React Native 以及 Xamarin，更重要的是它提供了完善的开发文档，以 Realm 的 React Native 文档 (<https://realm.io/docs/react-native/latest/>) 为例，从开始安装到常用 API 以及更高级的加密都有详细介绍和示例。

想要使用 Realm，首先还是添加 Realm 到 React Native 项目中，命令如下：

```
npm install realm --save          // 安装第三方库
react-native link realm           // 添加项目依赖
```

然后，使用 `react-native run-ios` 或 `react-native run-android` 命令重新编译和运行应用。

 **提示：** 第一次运行依赖 Realm 的 React Native 项目时，会下载 Realm 相关的包，由于国内网络环境的原因，下载速度可能较慢，因此需要耐心等待，或者自行搜索加速下载的其他办法。

重新编译和运行应用成功后，首先创建 Realm 的实例，创建 Realm 实例时，需要描述数据对象的名称、属性等信息，创建成功后，使用该对象就可以进行数据存储操作了，修改 `home.js` 代码如下：

```
01 // 这里省略了没有修改的代码
02 import Realm from 'realm';
03
04 export default class home extends Component {
05   constructor(props) {
06     super(props);
07     this.state = {
08       // 这里省略了没有修改的代码
09       realm: new Realm({                                // 创建 Realm 实例
10         schema: [
11           {
12             name: 'Product',                             // 数据对象的名称
13             properties: {                                // 数据对象的属性
14               id: 'int',
15               title: 'string',
```

```

16             subTitle: 'string',
17             image: 'string'
18         }
19     }
20 ]
21 ))
22 };
23 }
24 // 这里省略了没有修改的代码
25 }

```

然后, 继续添加 Realm 保存和读取数据的接口如下:

```

01 export default class home extends Component {
02     // 这里省略了没有修改的代码
03
04     _saveProducts = (products) => {
05         this.state.realm.write(() => {           // 使用 Realm 保存商品数据
06             for (const i = 0; i < products.length; i++) {
07                 const product = products[i];
08                 this.state.realm.create('Product', {
09                     id: parseInt(product.id),
10                     title: product.title,
11                     subTitle: product.subTitle,
12                     image: product.image
13                 });
14             }
15         });
16     }
17
18     _queryProducts = () => {                       // 使用 Realm 读取商品数据
19         return this.state.realm.objects('Product');
20     }
21 }

```

最后, 使用 Realm 的数据操作接口替换之前使用 AsyncStorage 存储和读取数据的实现, 修改 home.js 代码如下:

```


01 export default class home extends Component {
02     // 这里省略了没有修改的代码
03
04     _fetchProducts = () => {
05         const req = new Request(SERVER_URL + PRODUCT_API, {method:
06             'GET'});
07         fetch(req).then((res) => {
08             return res.json();
09         }).then((result, done) => {
10             if (!done) {
11                 this._saveProducts(result);
12                 this.setState({isNetworkValid: true,
13                     products: result});
14             }
15         }).catch((err) => {
16             const products = this._queryProducts();
17             console.log('products: ' + JSON.stringify(products));
18             this.setState({isNetworkValid: false,
19                 products: products});
19         });

```

```
20      }  
21  
22      // 这里省略了没有修改的代码  
23  }
```

在设备和服务器网络运行正常的情况下，成功获取商品数据后，再关闭设备网络或服务器，重新刷新应用，此时数据仍然可以正常获取和显示。

至此就实现了使用 Realm 进行 React Native 应用数据存取的功能。当然，Realm 的功能和接口还有很多，感兴趣的读者可以参考官方文档（<https://realm.io/docs/react-native/latest/>）继续深入学习。

 **提示：**React Native 上还有一些其他比较优秀的数据存储方案，例如基于 Redux 生态圈的 `redux-persist`（<https://github.com/rt2zz/redux-persist>），由于 Redux 的内容已经超出了本书讨论的范围，所以读者在熟练 React Native 基本开发和技巧之后，可以自行继续学习探索。

6.7 小 结

截止本章，我们不仅掌握了 React Native 各种原生组件、第三方组件，还为应用添加了完整的网络交互和数据存储的能力，这些内容已经可以满足一个 React Native 应用的基本开发需求。

但是需要再次提醒读者的是除了阅读本书，读者还需要反复地编码和练习，才能做到从学习知识到掌握知识的蜕变。

在熟练掌握这些 React Native 最基础也是最核心的内容后，我们将从第7章起，进入 React Native 和原生开发的“边界”，用 React Native 提供的 API 为应用添加更多功能。

第 7 章 常用 React Native API

在前面的章节中，我们开发了一个完整的 React Native 应用，熟悉了各种组件的用法，并且还使用了很多 React Native 提供的 API，包括

- Alert: 跨平台的提示框。
- AppRegistry: 注册 React Native 应用的入口。
- AsyncStorage: React Native 提供的键-值存储系统。
- Dimensions: 用于获取设备的屏幕宽高。
- Platform: 用于获取当前运行的平台名称。
- StyleSheet: 提供了一种类似 CSS 样式表的抽象。
- 定时器: setInterval/clearInterval 创建和销毁定时器。

那么，除了上述 API 之外，React Native 平台还有哪些常用的 API 呢？React Native API 与组件和原生接口又有什么关系呢？带着这些疑问开始本章的学习 React Native API。

本章主要内容有：

- 屏幕设置相关 API。
- 动画 API。
- 定义自己的 API。
- 更丰富的特定平台使用的 API，如 AlertIOS、AppState 等。

7.1 屏幕设置相关 API

在开发轮播效果时：

- 为了让轮播广告页面的宽度占满屏幕，使用了 Dimensions 来获取屏幕宽度。
- 为了让轮播广告每隔一段时间间隔滑动到下一页面，使用了定时器来定时更新页面。


后来在第三方组件的介绍中，又使用了第三方的轮播组件 react-native-swiper，但是该组件的实现原理仍然都是基于上述 API。

下面同样通过几个自定义组件或模块的例子，来进一步了解 React Native API 在实际应用开发过程中的能力和作用。

首先仍然是先创建 React Native 项目并安装依赖包：

```
react-native init ch06 // 新建 React Native 项目 ch06
cd ch06
npm install
```

然后将 ch04 项目中如下目录或文件都复制到 ch06 文件夹中。

提示：这里的 ch04 项目指未使用第三方组件的代码。如果读者按照第三方组件的方法使用 git 对 ch04 项目代码进行版本控制的话，可以先使用 `git checkout ch04-without-third-party-library` 命令恢复到使用第三方组件之前的 ch04 项目代码；如果读者没有使用 git 进行版本控制的话，也可以手动编辑和恢复 ch04 项目代码。

- app.js;
- detail.js;
- home.js;
- images;
- index.android.js;
- index.ios.js;
- main.android.js;
- main.ios.js;
- more.js。

另外，还需要修改 `index.ios.js` 和 `index.android.js` 中的代码如下：

```
01 import React, {Component} from 'react';
02 import {AppRegistry} from 'react-native';
03 import app from './app';
04
05 AppRegistry.registerComponent('ch06', () => app); // 修改第一个参数为'ch06'
```

此时，使用命令 `react-native run-ios` 或者 `react-native run-android` 运行 ch06 项目，确保 ch04 项目的实现成功移植到 ch06 项目中，效果如图 7.1 所示。



图 7.1 ch06 项目运行效果

7.1.1 获取屏幕宽高——Dimensions API

在保证 ch06 项目运行正常之后，先以获取手机屏幕分辨率为例，来看下更多 React

Native API 的用法。首先介绍 Dimensions API。

(1) 新建一个文件 Screen.js, 用于提供屏幕宽高的接口, 其中, 获取屏幕宽高仍然基于 Dimensions API。添加 Screen.js 代码如下:

```
01 import {Dimensions} from 'react-native';
02
03 export default {
04   'width' : Dimensions.get('window').width,
05   'height' : Dimensions.get('window').height
06 }
```

提示: 这里的 Screen.js 文件中导出的是一个普通的模块, 而不是类似 home.js 文件使用 extends React.Component 声明的 React Native 组件。

(2) 使用 Screen.js 的实现替换掉 home.js 文件中的 Dimensions 相关代码。修改 home.js 代码如下:

```
01 // 这里省略了没有修改的代码
02
03 import Detail from './detail';
04 import Screen from './Screen';
05
06 export default class home extends Component {
07   // 这里省略了没有修改的代码
08
09   render() {
10     const advertisementCount = this.state.advertisements.length;
11     const indicatorWidth = circleSize * advertisementCount +
12       circleMargin * advertisementCount * 2;
13     const left = (Screen.width - indicatorWidth) / 2;
14                                     // 使用 Screen 替换 Dimensions
15
16     return (
17       // 这里省略了没有修改的代码
18     );
19   }
20
21   // 这里省略了没有修改的代码
22
23   _startTimer() {
24     this.interval = setInterval(() => {
25       nextPage = this.state.currentPage + 1;
26       if (nextPage >= 3) {
27         nextPage = 0;
28       }
29
30       this.setState({currentPage: nextPage});
31
32       const offSetX = nextPage * Screen.width;
33                                     // 使用 Screen 替换 Dimensions
34       this.refs.scrollView.scrollResponderScrollTo({x: offSetX,
35         y: 0, animated: true});
36     }, 2000);
37   }
38
39   // 这里省略了没有修改的代码
40 }
41
42 const styles = StyleSheet.create({
43   // 这里省略了没有修改的代码
```

```

39     advertisementContent: {
40       width: Screen.width,
41       height: 180
42     },
43     // 这里省略了没有修改的代码
44   }

```

此时重新加载应用，发现首页没有任何变化，说明使用 Screen 替换 Dimensions 成功。

(3) 使用“更多”页面来展示实际的屏幕宽高。修改 more.js 代码如下：

```

01 import React, {Component} from 'react';
02 import {StyleSheet, View, Text} from 'react-native';
03
04 import Screen from '../Screen';
05
06 export default class more extends Component {
07   render() {
08     return (
09       <View style={styles.container}>
10         <Text style={styles.text}>width: {Screen.width}</Text>
11         <Text style={styles.text}>height: {Screen.height}</Text>
12       </View>
13     );
14   }
15 }
16
17 const styles = StyleSheet.create({
18   container: {
19     flex: 1,
20     justifyContent: 'center',
21     alignItems: 'center'
22   },
23   text: {
24     fontSize: 30
25   }
26 });

```

重新加载应用，打开“更多”页面，可以看到屏幕的宽高分别为 375 和 667（这里的设备为 iPhone 7），效果如图 7.2 所示。

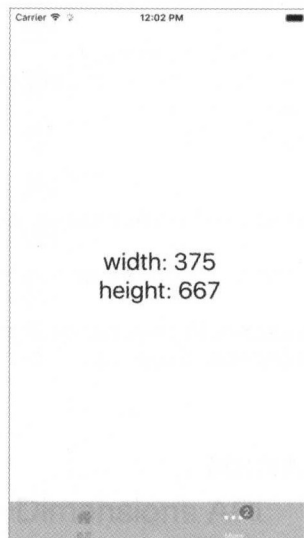


图 7.2 获取设备的屏幕宽高

🔔小³知识：所有 iPhone 屏幕的详细尺寸和分辨率，可以参考该网站的汇总 The Ultimate Guide To iPhone Resolutions (<https://www.paintcodeapp.com/news/ultimate-guide-to-iphone-resolutions>) 。

7.1.2 获取屏幕分辨率——PixelRatio API

获取屏幕的宽高，我们已经知道使用 Dimensions API。那么，如何获取屏幕的分辨率呢？

🔔提示：屏幕尺寸、屏幕分辨率以及屏幕像素密度的详细介绍，可以参考第 5 章内容，其中，屏幕尺寸是指手机屏幕对角线的英寸数；屏幕分辨率是指屏幕宽高像素数；屏幕像素密度是指手机屏幕对角线上单位英寸内的像素数。

这里需要用到一个新的 React Native API：PixelRatio，用于获取屏幕缩放比例。通过屏幕宽高和缩放比例计算屏幕分辨率的公式如下：

屏幕分辨率 = 屏幕宽高 × 屏幕缩放比例

其中，常见设备的屏幕缩放比例如表 7.1 所示。

表 7.1 常见设备的屏幕缩放比例

| 屏幕缩放比例 | 设 备 |
|--------------------------|---|
| PixelRatio.get() === 1 | mdpi Android设备 |
| PixelRatio.get() === 1.5 | hdpi Android设备 |
| PixelRatio.get() === 2 | iPhone 4/4S, iPhone 5/5c/5s, iPhone 6/6s, iPhone 7以及xxdpi Android设备 |
| PixelRatio.get() === 3 | iPhone 6 Plus, iPhone 6s Plus, iPhone 7 Plus以及xxhdpi Android设备 |
| PixelRatio.get() === 3.5 | Nexus 6 |

(1) 按照上述公式修改 Screen.js 代码如下：

```
01 import {Dimensions, PixelRatio} from 'react-native';
02
03 export default {
04   'width' : Dimensions.get('window').width,
05   'height' : Dimensions.get('window').height,
06   'pixelRatio' : PixelRatio.get(),
07   'resolutionX' : Dimensions.get('window').width * PixelRatio.get(),
08   'resolutionY' : Dimensions.get('window').height * PixelRatio.get()
09 }
```

(2) 在“更多”页面显示实际的屏幕分辨率，修改 more.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class more extends Component {
04   render() {
05     return (
06       <View style={styles.container}>
07         <Text style={styles.text}>width: {Screen.width}</Text>
08         <Text style={styles.text}>height: {Screen.height}</Text>
09         <Text style={styles.text}>pixelRatio: {Screen.pixelRatio}</Text>
10         <Text style={styles.text}>resolutionX: {Screen.resolutionX}</Text>
11       </View>
12     );
13   }
14 }
```

```
11         </Text>
12         <Text style={styles.text}>resolutionY: {Screen.resolutionY}</Text>
13     </View>
14   };
15 }
16
17 // 这里省略了没有修改的代码
```

(3) 重新加载应用，打开“更多”页面，可以看到屏幕缩放比例为2，屏幕分辨率为 $750=375\times 2$ 、 $1334=667\times 2$ （这里的设备为 iPhone 7），效果如图 7.3 所示。

(4) 再验证一下 Screen 在 Android 设备上的运行效果（这里的设备为 Nexus 5，属于 xxhdpi Android 设备），效果如图 7.4 所示。



图 7.3 iOS 设备屏幕缩放比例和屏幕分辨率

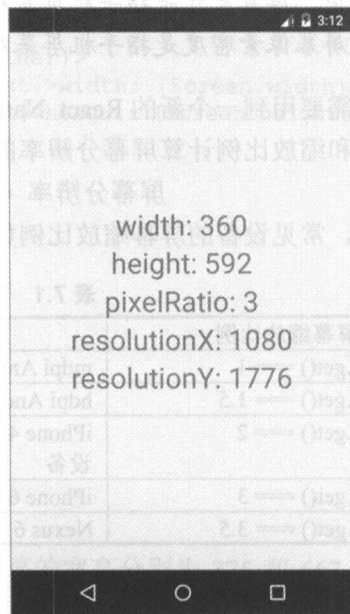


图 7.4 Android 设备屏幕缩放比例和屏幕分辨率

7.2 动画 API

在现在的移动应用开发中，流畅、有意义的动画对于用户体验是非常必要的。不输原生平台的是，React Native 也提供了简洁、强大的动画 API。

- **requestAnimationFrame**: 这个是最简单也是最“粗暴”的动画 API，通过不断改变 state 的值，来实现组件的动画效果。
- **LayoutAnimated**: 体验和性能更好，适用于全局的动画配置，实现单个动画非常简洁方便。
- **Animated**: 最强大的动画 API，适用于实现灵活丰富的动画效果，例如多个动画的组合动画。

7.2.1 RequestAnimationFrame API 帧动画

首先使用最简单的 requestAnimationFrame API 来实现 React Native 动画。

(1) 新建 Animation.js 文件，用于封装带有动画效果的组件，添加代码如下：

```
01 import React, {Component} from 'react';
02 import {StyleSheet, View} from 'react-native';
03
04 export default class Animation extends Component {
05   constructor(props) {
06     super(props);
07     this.state = {
08       width: parseInt(this.props.width),
09       height: parseInt(this.props.height)
10     }
11   }
12
13   render() {
14     return (
15       <View style={[
16         styles.animation, {
17           width: this.state.width,
18           height: this.state.height
19         }
20       ]}></View>
21     );
22   }
23 }
24
25 const styles = StyleSheet.create({
26   animation: {
27     backgroundColor: 'red'
28   }
29 })
```

(2) 在“更多”页面中显示该组件，修改 more.js 代码如下：

```
01 import React, {Component} from 'react';
02 import {StyleSheet, View} from 'react-native';
03
04 import Screen from './Screen';
05 import Animation from './Animation';
06
07 export default class more extends Component {
08   render() {
09     return (
10       <View style={styles.container}>
11         <Animation width='100' height='100'></Animation>
12       </View>
13     );
14   }
15 }
16
17 // 这里省略了没有修改的代码
```

重新加载应用，打开“更多”页面，效果如图 7.5 所示。

(3) 在 Animation 组件中添加 startAnimation() 函数：更新 state 中 width 和 height 的值。

修改 Animation.js 代码如下：


```
01 // 这里省略了没有修改的代码
02
03 export default class Animation extends Component {
04   // 这里省略了没有修改的代码
05
06   componentDidMount() {
07     this._startAnimation();
08   }
09
10   _startAnimation = () => {
11     let count = 0;
12     while (++count < 100) {
13       requestAnimationFrame(() => {
14         this.setState({
15           width: this.state.width + 1,
16           height: this.state.height + 1
17         });
18       });
19     }
20   }
21 }
22
23 // 这里省略了没有修改的代码
```

重新加载应用，打开“更多”页面，可以看到此时 Animation 组件的大小发生了动态更新，效果如图 7.6 所示。

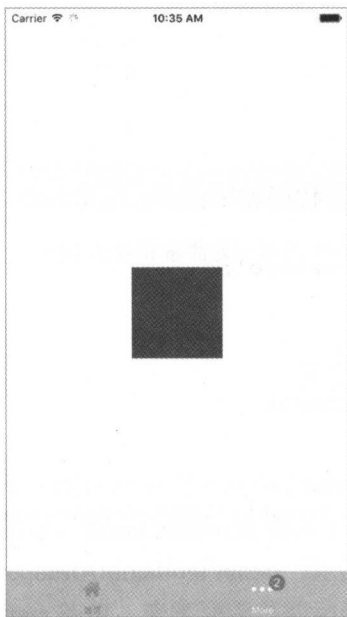


图 7.5 Animation 组件

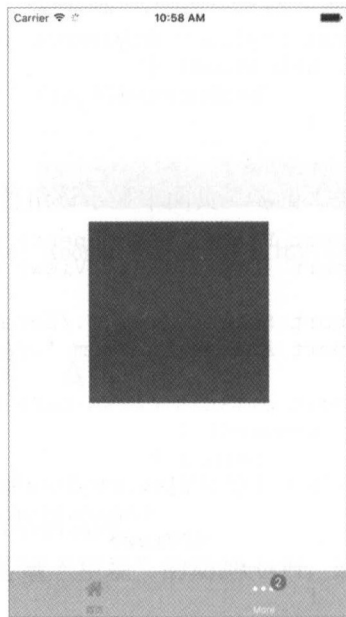


图 7.6 使用 requestAnimationFrame 实现的动画

不过细心的读者可能已经发现了，使用 requestAnimationFrame API 实现的动画效果比较生硬，如果想要实现“淡入淡出”或“弹性动画”等效果是比较困难的。因此，在实际的 React Native 开发中，通常还是使用以下两节介绍的动画 API。


7.2.2 LayoutAnimation API 布局动画

相比 `requestAnimationFrame`, `LayoutAnimation` API 要简单、智能得多: 当组件的布局变化时, 会自动将组件运行到新的位置上。

使用 `LayoutAnimation` API 的常用方法是调用 `LayoutAnimation.configureNext`, 然后使用 `setState` 设置组件的属性。

`configureNext` 函数用于配置动画效果, 可配置的选项如下。

- `duration`: 动画时长。
- `create`: 组件创建时的动画。
- `update`: 组件更新时的动画。
- `delete`: 组件销毁时的动画。

 提示: 读者也可以通过查看 React Native 源码的方式了解所有的配置选项, 例如, `LayoutAnimation` API 的 `configureNext()` 方法的配置就定义在 `node_modules/react-native/Libraries/LayoutAnimation/LayoutAnimation.js` 文件中。

`create`、`update` 以及 `delete` 动画的类型定义如下:

```
01 type Anim = {
02   duration?: number,           // 动画时长
03   delay?: number,             // 时间延迟, 单位: 毫秒
04   springDamping?: number,      // 弹跳动画阻尼系数, 配合 spring 使用
05   initialVelocity?: number,    // 初始速度
06   type?: $Enum<typeof TypesEnum>, // 动画类型
07   property?: $Enum<typeof PropertiesEnum>, // 动画属性
08 }
```

动画类型 `type` 定义在 `LayoutAnimation.Types` 中。

- `spring`: 弹跳。
- `linear`: 线性。
- `easeInEaseOut`: 缓入缓出。
- `easeIn`: 缓入。
- `easeOut`: 缓出。

动画属性 `property` 定义在 `LayoutAnimation.Properties` 中。

- `opacity`: 透明度。
- `scaleXY`: 缩放。

在了解完 `LayoutAnimation` 的配置方法之后, 下面来看一看使用 `LayoutAnimation` API 实现的动画效果。修改 `Animation.js` 代码如下:

```
01 // 这里省略了没有修改的代码
02
03 export default class Animation extends Component {
04   // 这里省略了没有修改的代码
05
06   _startAnimation = () => {
07     LayoutAnimation.configureNext({
```

```

08         duration: 1000, // 持续时间
09         create: { // 创建组件动画
10             type: LayoutAnimation.Types.spring, // 弹跳
11             property: LayoutAnimation.Properties.scaleXY, // 缩放
12         },
13         update: { // 更新组件动画
14             type: LayoutAnimation.Types.spring
15         }
16     });
17     this.setState({
18         width: this.state.width + 100,
19         height: this.state.height + 100
20     });
21 }
22 }
23
24 // 这里省略了没有修改的代码

```

重新加载应用，打开“更多”页面，可以看到此时的 Animation 组件有一个缩放效果的更新，结果如图 7.7 所示。

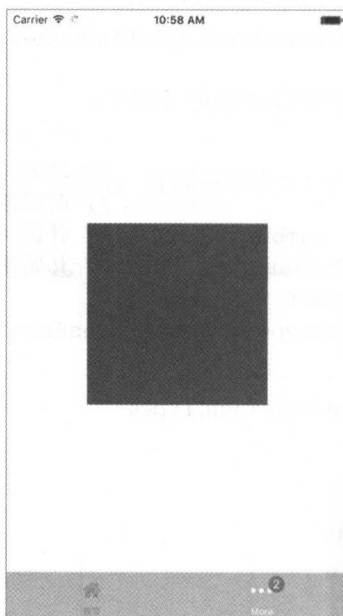




图 7.7 使用 LayoutAnimation 实现的动画

 **提示：**更多的动画配置和效果，限于篇幅这里就不一一展示了，读者可以自行尝试修改代码体验更多的动画效果。

想必读者已经体会到：使用 LayoutAnimation API 实现的动画效果非常流畅，而且动画的过程很柔和，丝毫没有生硬的感觉。同时，还可以灵活配置动画参数，已经能够基本满足单个动画的需求。但是，如果来实现复杂的组合动画，比如先缩小 50%，再向左平移 100 像素，那么就比较困难了，需要监听上一个动画的结束事件，再进行下一个动画。

 **提示：**configureNext() 第 2 个参数是可以监听动画结束的，以实现上述复杂的组合动画的需求，但是该方法暂时只在 iOS 平台有效。

那么,有没有一种更加灵活丰富的 API 呢? 答案就是下面要介绍的内容 Animated API。

7.2.3 Animated API 高级动画

相比 `LayoutAnimation` 全局的动画配置，`Animated` 使得开发者可以非常容易地实现各种各样的动画和交互，同时具备极高的性能。`Animated` 仅关注动画的输入与输出声明，在其中建立一个可配置的变化函数，然后使用简单的 `start()/stop()` 方法来控制动画按顺序执行。


使用 `Animated` 最简单的工作流程就是创建一个 `Animated.Value`，将其绑定到组件的一个或多个样式属性上。然后可以通过动画驱动它，例如 `Animated.timing`，或者通过 `Animated.event` 将其关联到一个手势上，例如拖动或者滑动操作。除了样式，`Animated.value` 还可以绑定到 `props` 上。

Animated 的动画类型有以下几种。

- spring: 弹跳。
- timing: 渐变。
- decay: 以一个初始速度开始并且逐渐减慢停止。

其中，spring 动画的配置选项定义如下：

```
01 type SpringAnimationConfig = AnimationConfig & {
02     toValue: number | AnimatedValue | {x: number, y: number} | Animated
    ValueXY,
03     overshootClamping?: bool,
04     restDisplacementThreshold?: number,
05     velocity?: number | {x: number, y: number},    // 初始速度
06     bounciness?: number,
07     speed?: number,                                // 速度
08     tension?: number,                              // 张力
09     friction?: number,                             // 摩擦系数
10 }
```

提示: 和上述 LayoutAnimation API 一样, 读者也可以通过查看 React Native 源码的方式了解所有的配置选项, 例如, Animated 动画的配置就定义在 node_modules/react-native/Libraries/Animated/src/AnimatedImplementation.js 文件中。

同理，timing 和 decay 动画的配置选项可以查看定义 `TimingAnimationConfig` 和 `DecayAnimationConfig`。

Animated 动画支持的组件有以下几种。

- `Animated.Text`;
- `Animated.Image`;
- `Animated.View`.

1. 弹跳动画

仍然以弹跳动画为例，下面来看看 `Animated` 的基本用法。修改 `Animation.js` 代码如下：

```
01 // 这里省略了没有修改的代码
02
```

```

03 export default class Animation extends Component {
04   constructor(props) {
05     super(props);
06     this.state = {
07       width: parseInt(this.props.width),
08       height: parseInt(this.props.height),
09       bounceValue: new Animated.Value(0) // 初始缩放为 0
10     }
11   }
12
13   render() {
14     return (
15       <Animated.View style={[ // 可选的基本组件类型: Image, Text, View
16         styles.animation, {
17           width: this.state.width,
18           height: this.state.height,
19           transform: [ // transform 是一个有序数组 (动画按顺序执行)
20             {
21               scale: this.state.bounceValue
22             }
23           ]
24         }
25       ]}></Animated.View>
26     );
27   }
28
29   componentDidMount() {
30     this._startAnimation();
31   }
32
33   _startAnimation = () => {
34     Animated.spring(this.state.bounceValue, {
35       // 可选的基本动画类型: spring, decay, timing
36       toValue: 1
37     }).start(); // 开始执行动画
38   }
39
40   // 这里省略了没有修改的代码

```

同时, 修改 `Animated` 组件的宽高为 `200×200`。修改 `more.js` 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 export default class more extends Component {
04   render() {
05     return (
06       <View style={styles.container}>
07         <Animation width='200' height='200'></Animation>
08       </View>
09     );
10   }
11 }
12
13 // 这里省略了没有修改的代码

```

重新加载应用, 打开“更多”页面, 可以看到弹跳的动画效果如图 7.8 和图 7.9 所示。

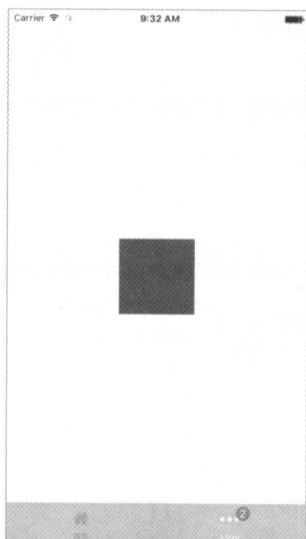


图 7.8 使用 Animated 实现的弹跳动画 1

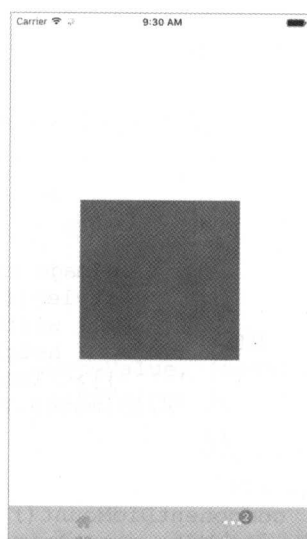


图 7.9 使用 Animated 实现的弹跳动画 2

2. 串行动画

当然，Animated 能做的动画远不止这么简单，相比 LayoutAnimation，Animated 还支持组合动画。

- sequence: 串行。
- parallel: 并行。
- stagger: 交错，其实就是插入了 delay 的 parallel。
- delay: 组合动画之间的延迟方法，严格来讲并不算是组合动画。

这么强大的动画 API，想必读者已经迫不及待要看下它的效果了。

首先，来看下串行组合动画的用法和效果，修改 Animation.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class Animation extends Component {
04   constructor(props) {
05     super(props);
06     this.state = {
07       width: parseInt(this.props.width),
08       height: parseInt(this.props.height),
09       bounceValue: new Animated.Value(0),
10       rotateValue: new Animated.Value(0)
11     }
12   }
13
14   render() {
15     return (
16       <Animated.View style={{
17         styles.animation, {
18           width: this.state.width,
19           height: this.state.height,
20           transform: [ // transform 是一个有序数组（动画按顺序执行）
21             {
22               scale: this.state.bounceValue
23             }, {
24               rotate: this.state.rotateValue.interpolate({
```

```

25         inputRange: [
26             0, 1
27         ],
28         outputRange: ['0deg', '360deg']
29     })
30 }
31 ]
32 }
33 ]}]>
34 <Image source={require('./images/product-image-01.jpg')}
35     style={{
36         width: this.state.width,
37         height: this.state.height
38     }}></Image>
39 </Animated.View>
40 );
41 }
42 componentDidMount() {
43     this._startAnimation();
44 }
45
46 _startAnimation = () => {
47     Animated.sequence([                // 串行动画
48         Animated.spring(this.state.bounceValue, {toValue: 1}),
49         Animated.delay(500),           // 串行动画延迟时间
50         Animated.timing(this.state.rotateValue, {
51             toValue: 1,
52             duration: 800,              // 持续时间, 默认为 500
53             easing: Easing.out(Easing.quad), // 渐变函数
54         })
55     ]).start();                        // 开始执行动画
56 }
57 }
58
59 // 这里省略了没有修改的代码

```

上述代码实现的动画效果是先弹出动画, 然后延迟 0.5 秒钟, 最后旋转动画。效果如图 7.10 和图 7.11 所示。



图 7.10 使用 Animated 实现的串行动画 1



图 7.11 使用 Animated 实现的串行动画 2

3. 并行动画

此外，还可以将串行动画改成并行动画。修改 Animation.js 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class Animation extends Component {
04   // 这里省略了没有修改的代码
05
06   _startAnimation = () => {
07     Animated.parallel([           // 并行动画
08       Animated.spring(this.state.bounceValue, {toValue: 1}),
09       Animated.timing(this.state.rotateValue, {
10         toValue: 1,
11         duration: 800,           // 持续时间，默认为 500
12         easing: Easing.out(Easing.quad), // 渐变函数
13       })
14     ]).start();                 // 开始执行动画
15   }
16 }
17
18 // 这里省略了没有修改的代码

```

重新加载应用，打开“更多”页面，可以看到并行动画效果如图 7.12 所示。



图 7.12 使用 Animated 实现的并行动画

至此，使用不同 React Native 动画 API 实现的 Animation 组件就呈现在读者的面前了。

提醒：动画效果没有一个固定的评判标准，因此实际开发中的动画效果是需要不断探索和调整的，在开发动画的过程中，开发人员需要和设计师、用户不断沟通、改进，这样才能获得更好的用户体验。

7.3 组件、React Native API、原生平台 API

在之前的章节中已经接触并使用了很多的 API 和组件，那么在 React Native 平台中，API 和组件到底有什么关系和区别呢？还有原生平台的 API 又是怎么回事？

7.3.1 组件和 API

在理清 API 与组件的关系和区别之前，必须先清楚什么是 API，什么是组件。

- API（Application Programming Interface）是指应用程序编程接口，在 React Native 平台上，API 是一些预先定义并实现好的函数，基于 React native 平台 API，应用开发者通过调用这些接口就可以达到预期的目的，而无须了解 React Native 内部工作的细节。例如，使用 Dimensions API 可以获取设备的屏幕宽高，我们并不关心获取的原理和过程。
- 组件（Component）是对数据和方法的简单封装，可以理解为一个组件就是一个对象，它可以有自己的属性和方法。React Native 应用中，所有展示的界面都可以看做是一个组件，它们只是功能和逻辑上的复杂程度不同。每一个组件都是由许多小的组件组合而成，每个小的组件也有自己对应的逻辑，不过它们都遵循同样的代码结构。例如前面封装的动画组件 Animation.js。

当然，在组件的内部也可能需要使用相关的 API。例如，滑动的轮播广告中实现的轮播广告，使用了 setInterval/clearInterval 定时器 API，以及动画中封装的动画组件 Animation，使用了 Animated API 来实现动画效果。

综上所述，在 React Native 平台中，API 和组件的关系及结构如图 7.13 所示。

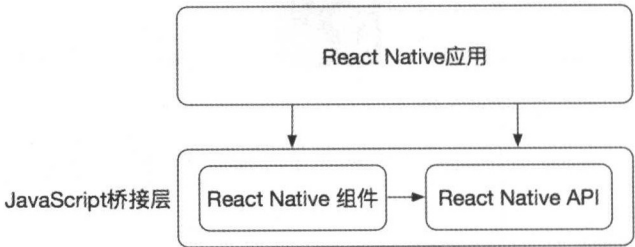


图 7.13 React Native 中 API 和组件的关系和结构

7.3.2 API 和原生平台 API

原生平台 API 是 iOS 或 Android 本身的 API，那么 React Native API 和原生平台又是怎么交互的？带着这些疑问，我们还是以 Animated API 为例来看看能否找到答案。

首先，打开 React Native 中 Animated API 的相关源码 node_modules/react-native/Libraries/Animated/src/NativeAnimatedHelper.js 文件，该文件代码如下：

```

01 // 这里省略了无关的代码
02
03 const NativeAnimatedModule = require('NativeModules').NativeAnimated
  Module;
04 const NativeEventEmitter = require('NativeEventEmitter');
05
06 // 这里省略了无关的代码

```

可以看到, `NativeAnimatedHelper.js` 文件中导入了 `NativeModules` 和 `NativeEventEmitter` 模块, 那么这两个模块又是干什么用的呢?

- **NativeModules:** 用于 JavaScript 代码调用原生代码。
- **NativeEventEmitter:** 用于原生代码发送消息到 JavaScript 代码。

由此不难推断出: `Animated` API 通过 `NativeAnimatedHelper` 来与原生平台通信, 它的实现仍然是基于原生的。

通过 `Animated` API 的例子知道, 在 `React Native` 平台中, API 和原生接口的关系和结构如图 7.14 所示。

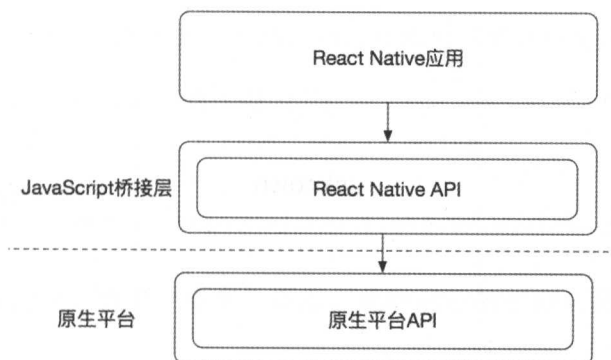


图 7.14 React Native 中 API 和原生接口的关系和结构

7.4 实现自己的 Platform API

为了进一步理解 `React Native` API 和原生接口的关系, 这里以 `Platform API` 为例, 实现一个相同功能的 API 来代替它。

(1) 新建 `Platform.js` 文件, 添加代码如下:

```

01 export default {
02   'systemName' : 'unknown'
03 }

```

(2) 在“更多”页面中显示 `Platform.systemName` 的值, 修改 `more.js` 代码如下:

```

01 // 这里省略了无关的代码
02
03 export default class more extends Component {
04   render() {
05     return (
06       <View style={styles.container}>
07         <Text style={styles.text}>{Platform.systemName}</Text>

```

```
08         </View>
09     );
10 }
11 }
12
13 const styles = StyleSheet.create({
14     // 这里省略了无关的代码
15     text: {
16         fontSize: 30
17     }
18 });
```

(3) 重新加载应用，打开“更多”页面，可以看到此时显示的平台名称为 unknown，效果如图 7.15 所示。



图 7.15 自定义 Platform API 初始效果

在准备好模块的定义和页面展示之后，就可以专心开发与原生平台交互的接口了。

7.4.1 支持 iOS 平台

支持 iOS 平台的接口设计步骤如下。

(1) 对于 iOS 平台，使用 Xcode 打开 ios/ch07.xcodeproj 文件，然后新建 Platform 类：生成两个文件 Platform.m 和 Platform.h，其中 Platform.m 是源文件，Platform.h 是头文件，效果如图 7.16 所示。

(2) 修改 Platform.h 代码如下：

```
01 #import <React/RCTBridgeModule.h>
02
03 @interface Platform : NSObject <RCTBridgeModule>
```

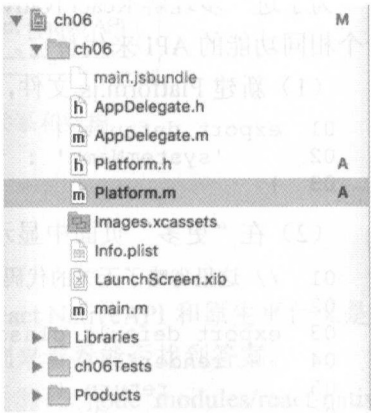


图 7.16 在 iOS 工程中新建 Platform 类

```
04
05 @end
```

其中，头文件 `RCTBridgeModule.h` 和 `RCTBridgeModule` 是 React Native 平台为开发者提供的与原生平台通信的桥梁实现。

(3) 修改源文件 `Platform.m` 代码如下：

```
01 #import "Platform.h"
02
03 @implementation Platform
04
05 RCT_EXPORT_MODULE()           // 导出此原生模块供 React Native 接口调用
06
07 - (NSDictionary *)constantsToExport {
08     return @{                  // 返回键值对
09         @"systemName": @"ios"
10     };
11 }
12
13 @end
```

 提示：关于原生接口和代码的更多介绍，将在后面内容中详细讨论。

(4) 最后修改 `Platform.js` 中的代码如下：

```
01 import {NativeModules} from 'react-native';
02
03 export default {
04     'systemName' : NativeModules.Platform.systemName
05 }
```

重新加载 iOS 应用，查看“更多”页面，此时显示的平台名称就是 ios 了，效果如图 7.17 所示。

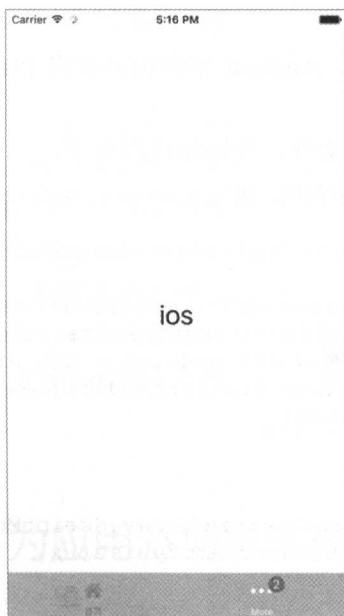


图 7.17 iOS 平台的自定义 Platform API

7.4.2 支持 Android 平台

对于 Android 平台,使用 Android Studio 打开 android 文件夹,然后新建 PlatformModule 类:生成文件 PlatformModule.java,效果如图 7.18 所示。

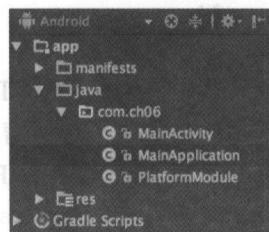


图 7.18 在 Android 工程中
新建 PlatformModule 类

(1) 修改 PlatformModule.java 代码如下:

```
01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class PlatformModule extends ReactContext.BaseJavaModule {
04     private final ReactApplicationContext mReactContext;
05
06     public PlatformModule(ReactApplicationContext reactContext) {
07         super(reactContext);
08         this.mReactContext = reactContext;
09     }
10
11     @Override
12     public String getName() {
13         return "Platform"; // 模块名称
14     }
15
16     @Override
17     public @Nullable
18     Map<String, Object> getConstants() {
19         HashMap<String, Object> constants = new HashMap<String, Object>();
20         constants.put("systemName", "android");
21         return constants; // 返回键值对
22     }
23 }
```

与 iOS 平台导出模块类似, Android 平台也需要将 PlatformModule 模块导出供 React Native 接口调用。

(2) 再新建 Platform.java 文件, 并添加代码如下:

```
01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class Platform implements ReactPackage {
04     @Override
05     public List<NativeModule> createNativeModules(
06         ReactApplicationContext reactContext) {
07         List<NativeModule> modules = new ArrayList<>();
08         modules.add(new PlatformModule(reactContext));
09         return modules;
10     }
11
12     @Override
13     public List<Class<? extends JavaScriptModule>> createJSModules() {
14         return Collections.emptyList();
15     }
16
17     @Override
18     public List<ViewManager> createViewManagers(
19         ReactApplicationContext reactContext) {
20         return Collections.emptyList();
21     }
22 }
```

```

21     }
22 }

```

(3) 接着还需要在 `MainApplication.java` 文件中注册，才算正式完成导出 `Platform` 模块的操作。修改 `MainApplication.java` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 public class MainApplication extends Application implements React
    Application {
04
05     private final ReactNativeHost mReactNativeHost = new ReactNative
        Host(this) {
06         // 这里省略了没有修改的代码
07
08         @Override
09         protected List<ReactPackage> getPackages() {
10             return Arrays.<ReactPackage>asList(
11                 new MainReactPackage(), new Platform()
                                // 导出 Platform 模块
12             );
13         }
14     };
15
16     // 这里省略了没有修改的代码
17 }

```

重新编译和运行 `Android` 应用，查看“更多”页面，此时显示的平台名称就是 `android` 了，效果如图 7.19 所示。

至此，一个自定义的跨平台 `Platform API` 就实现了。从这个例子中可以进一步了解到 `React Native API` 与原生接口的关系：`React Native API` 基于原生平台接口。

提示：本节内容涉及了原生平台开发的部分知识，例如 `Xcode` 和 `Android Studio` 工具的使用，以及 `Objective C` 和 `Java` 编程语言，限于篇幅，本书没有详细介绍。但是请读者不用担心，按照本书代码示例即可完成开发。当然，如果读者感兴趣的话，也可以参考 `iOS/Android` 开发的相关书籍和教程。

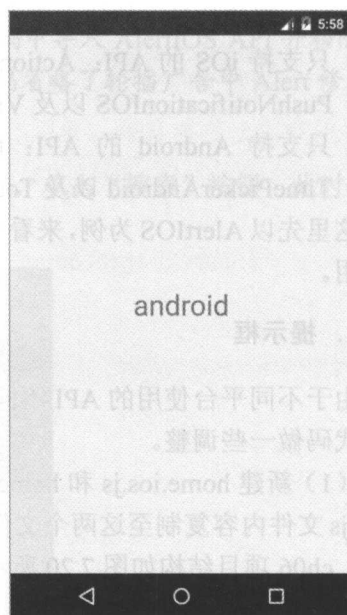


图 7.19 `Android` 平台的自定义 `Platform API`

7.5 为应用添加更丰富的 API

随着应用的功能愈加完善，所用到的 `React Native API` 也越来越多，目前在电商 App 中已经使用的 `API` 有：

- Alert;
- Animated;
- AppRegistry;
- AsyncStorage;
- Dimensions;
- Easing;
- LayoutAnimation;
- PixelRatio;
- Platform;
- StyleSheet;
- 定时器。

不过这些只是 React Native 庞大 API 的冰山一角，因此下面再介绍一些其他比较常用的 API。

7.5.1 提示框和编辑框——AlertIOS

迄今，我们使用的都是跨平台 API，即同时支持 iOS 和 Android 平台的 API，而还有一类 API，那就是只支持特定平台的 API，例如


- 只支持 iOS 的 API: ActionSheetIOS、AdSupportIOS、AlertIOS、ImagePickerIOS、PushNotificationIOS 以及 VibrationIOS 等。
- 只支持 Android 的 API: BackAndroid、DatePickerAndroid、PermissionsAndroid、TimePickerAndroid 以及 ToastAndroid 等。

这里先以 AlertIOS 为例，来看一看只支持特定平台 API 的使用。

1. 提示框

由于不同平台使用的 API 不同，这里首先要对 ch06 项目的代码做一些调整。

(1) 新建 home.ios.js 和 home.android.js 两个文件，将 home.js 文件内容复制至这两个文件中后删除 home.js 文件，此时，ch06 项目结构如图 7.20 所示。

 **提示：**关于 home.ios.js 和 home.android.js 跨平台代码的原理和机制，读者可以参考 4.1.1 节的内容。

(2) 使用 AlertIOS API 替换 home.ios.js 原有的 Alert API。修改 home.ios.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04   // 这里省略了没有修改的代码
05 }
```

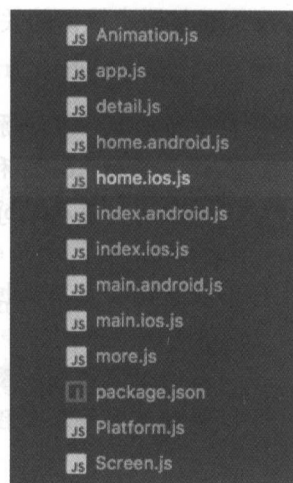



图 7.20 重构后的 ch06 项目结构

```

06   render() {
07     // 这里省略了没有修改的代码
08     return (
09       <View style={styles.container}>
10         // 这里省略了没有修改的代码
11         <View style={styles.searchbar}>
12           <TextInput style={styles.input}
13             placeholder='搜索商品'
14             onChangeText={(text) => {
15               this.setState({searchText: text});
16             }}></TextInput>
17           <Button style={styles.button}
18             title='搜索'
19             onPress={() => {
20               AlertIOS.alert('搜索内容: ' + this.state.
21                 searchText, null, null);
22             }}></Button>
23         </View>
24         // 这里省略了没有修改的代码
25       </View>
26     );
27   }
28   // 这里省略了没有修改的代码
29 }

```

 **注意：**使用 AlertIOS API 替换 Alert API，需要在代码中导入 AlertIOS API 并移除 Alert API。另外，因为与搜索框完全相同，上述代码省略了轮播广告中 Alert 修改的说明，请读者知悉。

(3) 重新加载应用，输入要搜索的内容，例如 Abc，单击“搜索”按钮，此时提示框将显示输入框刚才输入的 Abc，效果如图 7.21 所示。

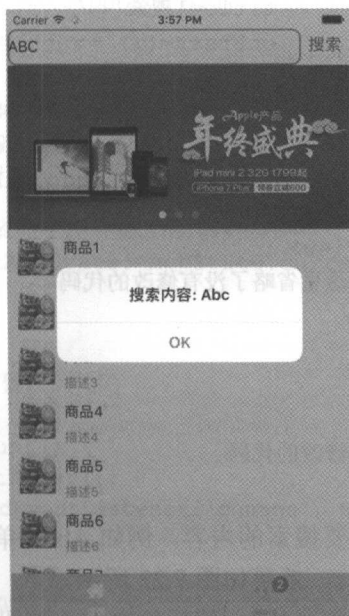


图 7.21 基于 AlertIOS 的提示框

2. 编辑框

细心的读者可能会发现：使用 AlertIOS 的提示框和 Alert 提示框没有什么区别，而且 Alert API 又是跨平台的，那么到底 AlertIOS API 还有什么用呢？

- 如果仅仅显示一个静态的提示框，应该使用跨平台的 Alert API。
- 如果针对 iOS 平台需要提示用户输入一些信息的话，可以使用 AlertIOS API。

下面以编辑搜索结果的功能为例，来看看 AlertIOS API 的这个“特长”。

(1) 修改 home.ios.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04   // 这里省略了没有修改的代码
05
06   render() {
07     // 这里省略了没有修改的代码
08     return (
09       <View style={styles.container}>
10         // 这里省略了没有修改的代码
11         <View style={styles.searchbar}>
12           <TextInput style={styles.input}
13             placeholder='搜索商品'
14             value={this.state.searchText}
15             onChangeText={(text) => {
16               this.setState({searchText: text});
17             }}></TextInput>
18           <Button style={styles.button}
19             title='搜索'
20             onPress={() => {
21               AlertIOS.prompt('编辑搜索结果', null,
22                 (promptValue) => {
23                   this.setState({searchText: promptValue});
24                 }, undefined, this.state.searchText);
25             }}></Button>
26           // 这里省略了没有修改的代码
27         </View>
28       )
29     };
30
31     // 这里省略了没有修改的代码
32 }
```

(2) 重新加载应用，输入要搜索的内容，例如 Abc，单击“搜索”按钮，此时会弹出提示框编辑输入刚才输入的 Abc，效果如图 7.22 所示。

(3) 接着在编辑器框中输入 Abcd，然后单击 OK 按钮，此时搜索输入框中的搜索结果被修改了，效果如图 7.23 所示。

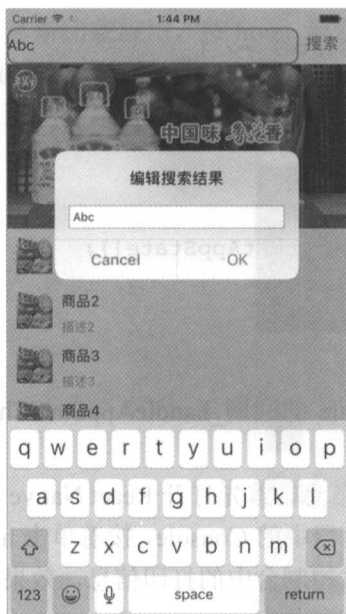


图 7.22 基于 AlertIOS 的编辑框



图 7.23 更新搜索输入框

7.5.2 前后台状态变化——AppState

AppState API 能告诉开发者应用当前是在前台还是在后台，并且能在状态变化的时候发出通知。基于 AppState API，可以在应用从后台进入前台的时候，自动刷新列表的数据，以达到更好的用户体验。

(1) 在 home.ios.js 文件中导入 AppState API，修改 home.ios.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04   constructor(props) {
05     super(props);
06     this.state = {
07       // 这里省略了没有修改的代码
08       currentAppState: AppState.currentState
09     }
10   }
11
12   // 这里省略了没有修改的代码
13
14   componentDidMount() {
15     this._startTimer();
16     AppState.addEventListener('change', this._handleAppStateChange);
17   }
18
19   componentWillUnmount() {
20     clearInterval(this.interval);
21     AppState.removeEventListener('change', this._handleAppStateChange);
22   }
```

```

23
24   _handleAppStateChange = (nextAppState) => {
25     if (nextAppState === 'inactive' || nextAppState === 'background') {
26       console.log('应用进入后台');
27     } else if (nextAppState === 'active') {
28       console.log('应用进入前台');
29     }
30
31     this.setState({currentAppState: nextAppState});
32   }
33
34   // 这里省略了没有修改的代码
35 }

```

在 `componentDidMount()` 函数中, 注册了 `AppState` 的回调 `_handleAppStateChange`, 当 `AppState` 发生 `change` 事件时, 会调用注册的回调。

(2) 重新加载应用后, 为了查看回调中的打印信息, 首先打开 React Native 调试选项中的 `Debug JS Remotely` 选项, 然后选择 Chrome 浏览器中的 `Console`, 接着单击 `home` 键返回桌面, 最后重新打开应用。此时, 就可以看到调试控制台中的打印信息, 效果如图 7.24 所示。

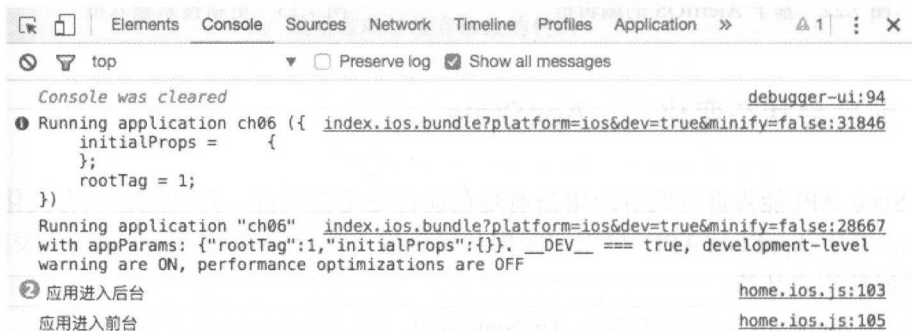


图 7.24 React Native 调试的终端控制台

(3) 在导入和理解了 `AppState` API 的使用之后, 就可以进一步优化代码: 当应用从后台进入前台的时候, 获取和刷新商品列表的数据。修改 `home.ios.js` 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04   // 这里省略了没有修改的代码
05
06   _handleAppStateChange = (nextAppState) => {
07     if (nextAppState === 'active') {
08       if (this.state.currentAppState === 'inactive'
09         || this.state.currentAppState === 'background') {
10         this._onRefresh();
11       }
12
13       this.setState({currentAppState: nextAppState});
14     }
15
16     // 这里省略了没有修改的代码
17 }

```

(4) 重新加载应用，按 home 键返回桌面，然后重新打开应用。此时商品列表的数据自动重新刷新，效果如图 7.25 所示。



图 7.25 应用从后台进入前台时自动刷新商品列表

7.5.3 Android 物理“返回键”——BackAndroid

在 7.5.1 节中，我们了解了只支持 iOS 平台的 API，下面将介绍一个只支持 Android 平台的 API: BackAndroid。

众所周知，iOS 设备是没有物理返回按键的，而 Android 设备通常是有返回按键的。React Native 提供的 BackAndroid API 就是用来处理返回按键的。修改 home.android.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04   // 这里省略了没有修改的代码
05
06   componentDidMount() {
07     this._startTimer();
08     AppState.addEventListener('change', this._handleAppStateChange);
09     if (Platform.OS === 'android') {
10       BackAndroid.addEventListener('hardwareBackPress', this._
         onBackAndroid);
11     }
12   }
13
14   componentWillUnmount() {
15     clearInterval(this.interval);
16     AppState.removeEventListener('change', this._handleAppStateChange);
17     if (Platform.OS === 'android') {
```

```

18         BackAndroid.removeEventListener('hardwareBackPress',
19           this._onBackAndroid);
20       }
21     }
22     _onBackAndroid = () => {
23       Alert.alert('点击返回按钮', null, null);
24       return true;
25     }
26
27     // 这里省略了没有修改的代码
28   }

```

在 Android 设备上，当按“返回”按键时，如果没有监听 `hardwareBackPress` 事件的回调或者回调返回的值不为 `true`，那么返回事件会按照系统默认的方式进行处理。而在上述代码中，我们注册了回调，该回调弹出提示框后返回 `true`，因此当按“返回”按键时，应用并不会返回到桌面，效果如图 7.26 所示。

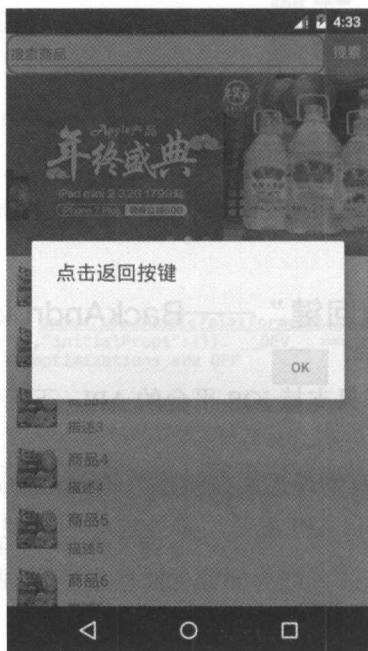


图 7.26 使用 BackAndroid 处理“返回”按钮

7.5.4 日期和时间选择器——DatePickerAndroid/TimePickerAndroid

`DatePickerAndroid` 和 `TimePickerAndroid` 都是只支持 Android 平台的 API。其中：

- `DatePickerAndroid` API 会打开一个标准的 Android 日期选择器的对话框。
- `TimePickerAndroid` API 会打开一个标准的 Android 时间选择器的对话框。

同样，由于不同平台使用的 API 不同，这里首先要对 `ch06` 项目的代码做一些调整。新建 `more.ios.js` 和 `more.android.js` 两个文件，将 `more.js` 文件内容复制至这两个文件后删除 `more.js` 文件，此时 `ch06` 项目结构如图 7.27 所示。

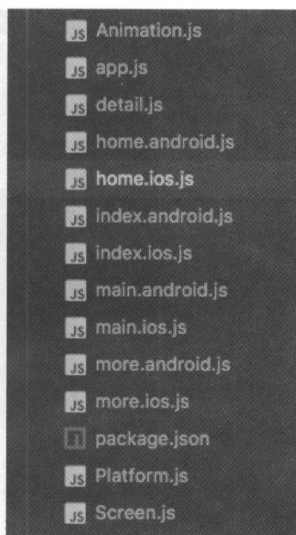


图 7.27 重构后的 ch06 项目结构

1. DatePickerAndroid 日期选择器

首先, 在“更多”页面添加显示日期的文本和“日期选择器”按钮。修改 `more.android.js` 代码如下:

```
01 // 这里省略了没有修改的代码
02
03 export default class more extends Component {
04   constructor(props) {
05     super(props);
06     this.state = {
07       dateText: '选择一个日期'
08     }
09   }
10
11   render() {
12     return (
13       <View style={styles.container}>
14         <Text style={styles.text}>{this.state.dateText}</Text>
15         <Button title='日期选择器' onPress={() => {
16           console.log('单击日期选择按钮');
17         }}></Button>
18       </View>
19     );
20   }
21
22 // 这里省略了没有修改的代码
```

重新加载应用, 打开“更多”页面, 可以看到添加的文本和按钮效果如图 7.28 所示。



图 7.28 添加文本和“日期选择器”按钮

接着，完善“日期选择器”按钮的单击响应事件，修改 `more.android.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class more extends Component {
04   // 这里省略了没有修改的代码
05
06   render() {
07     return (
08       <View style={styles.container}>
09         <Text style={styles.text}>{this.state.dateText}</Text>
10         <Button title='日期选择器' onPress={this._showDatePicker}>
11           </Button>
12       </View>
13     );
14   }
15
16   _showDatePicker = async() => { // 因为 await 是异步的，所以这里使用 async
17     try {
18       let newState = {};
19       const {action, year, month, day} = await DatePickerAndroid.
20         open(); // 使用 await 等待选择完处理结果
21       if (action === DatePickerAndroid.dismissedAction) {
22         newState['dateText'] = '取消选择';
23       } else {
24         let date = new Date(year, month, day);
25         newState['dateText'] = date.toLocaleDateString();
26       }
27       this.setState(newState);
28     } catch ({code, message}) {
29       console.warn('打开 DatePickerAndroid 失败: ' + message);
30     }
31   }

```

32

33 // 这里省略了没有修改的代码

重新加载应用，在“更多”页面，单击“日期选择器”按钮，弹出日期选择器，选择某一日期后单击 OK 按钮，效果如图 7.29 和图 7.30 所示。



图 7.29 DatePickerAndroid 日期选择器

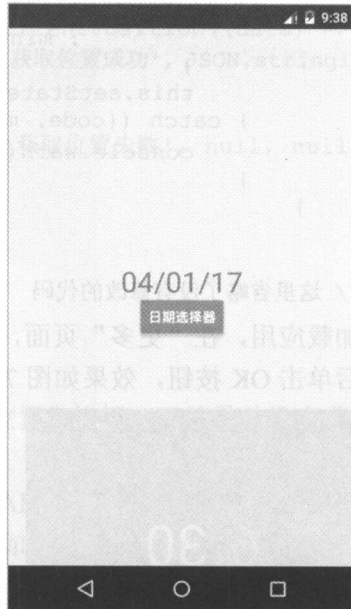


图 7.30 选择具体日期

2. TimePickerAndroid 时间选择器

和 DatePickerAndroid 类似，在“更多”页面添加显示时间的文本和“时间选择器”按钮。修改 more.android.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class more extends Component {
04   constructor(props) {
05     super(props);
06     this.state = {
07       timeText: '选择一个时间'
08     }
09   }
10
11   render() {
12     return (
13       <View style={styles.container}>
14         <Text style={styles.text}>{this.state.timeText}</Text>
15         <Button title='时间选择器' onPress={this._showTimePicker}>
16           </Button>
17       </View>
18     );
19   }
20
21   _showTimePicker = async() => {
22     try {
23       let newState = {};
```

```

23         const {action, minute, hour} = await TimePickerAndroid.
          open();
24         if (action === TimePickerAndroid.dismissedAction) {
25             newState['timeText'] = '取消选择';
26         } else {
27             newState['timeText'] = hour + ':' + (minute < 10
28                 ? '0' + minute
29                 : minute);
30         }
31         this.setState(newState);
32     } catch ({code, message}) {
33         console.warn('打开 TimePickerAndroid 失败: ' + message);
34     }
35 }
36 }
37
38 // 这里省略了没有修改的代码

```

重新加载应用，在“更多”页面，单击“时间选择器”按钮，弹出时间选择器，选择某一时间后单击 OK 按钮，效果如图 7.31 和图 7.32 所示。

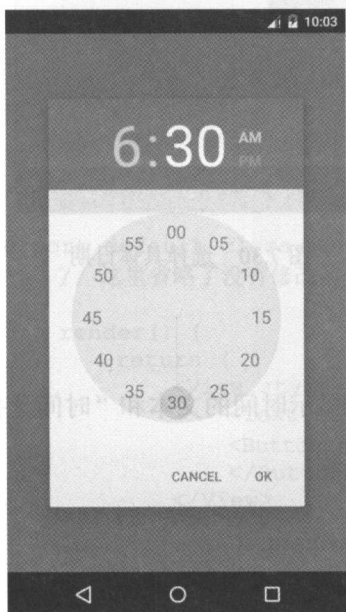


图 7.31 TimePickerAndroid 时间选择器

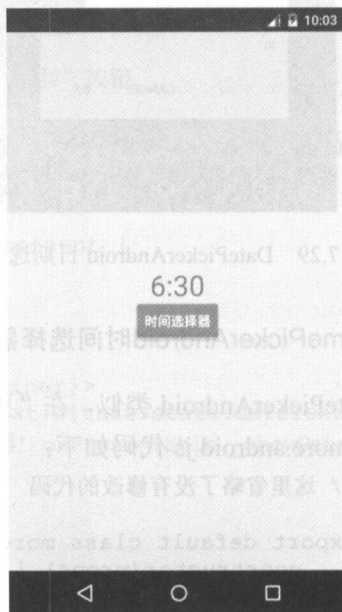


图 7.32 选择具体时间

7.5.5 基于位置的 Geolocation

基于位置的服务已经成为应用中必不可少的功能，本书前面曾介绍过地图组件的使用，这里将介绍 React Native 提供的地理位置 API: Geolocation。

下面直接用一个例子来展示 Geolocation API 的用法。

1. Android平台

对于 Android 平台，修改 more.android.js 代码如下：

```

01 import React, {Component} from 'react';
02 import {StyleSheet, View, Button, Alert} from 'react-native';

```

```

03 import Geolocation from 'Geolocation';
04
05 export default class more extends Component {
06   render() {
07     return (
08       <View style={styles.container}>
09         <Button title='获取位置' onPress={() => {
10           Geolocation.getCurrentPosition((data) => {
11             Alert.alert('获取位置成功', JSON.stringify(data),
12               null);
13             }, () => {
14               Alert.alert('获取位置失败', null, null);
15             })
16           }}></Button>
17       </View>
18     );
19   }
20 }
21 // 这里省略了没有修改的代码

```

重新加载应用，在“更多”页面中单击“获取位置”按钮，可结果并没有获取到位置，而是发生了如图 7.33 所示的错误。

仔细阅读错误信息可以发现，使用 Geolocation API 要请求访问地理位置的权限，于是，在 Android 工程的 AndroidManifest.xml 文件中添加如下配置：

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

📌 注意：通常在修改原生项目工程中的文件后（例如上述的 AndroidManifest.xml 文件），需要使用 react-native run-ios 或 react-native run-android 命令重新编译和安装 React Native 应用。

重新加载应用，单击“更多”页面中的“获取位置”按钮，此时可以获取到当前的位置信息，如图 7.34 所示。

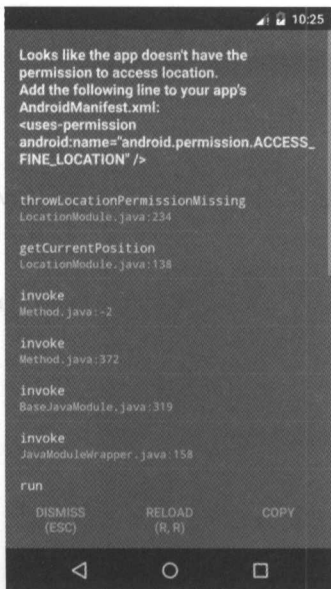


图 7.33 使用 Geolocation API 发生的权限错误

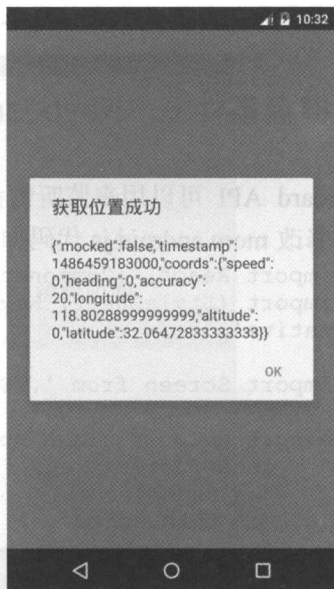


图 7.34 使用 Geolocation API 获取位置信息

2. iOS平台

对于 iOS 平台，直接复制 `more.android.js` 文件中的代码至 `more.ios.js` 文件中，然后重新加载应用，单击“更多”页面中的“获取位置”按钮，就可以获取到当前的位置信息，如图 7.35 所示。

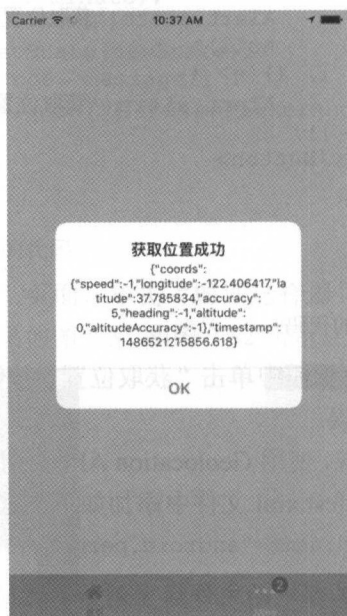



图 7.35 iOS 平台使用 Geolocation API 获取位置信息

 提示：iOS 平台使用 Geolocation API 也是需要请求权限的，在 iOS 工程的 `Info.plist` 文件中增加 `NSLocationWhenInUseUsageDescription` 字段。如果使用 `react-native init` 创建项目，定位功能会被默认启用。

7.5.6 键盘事件——Keyboard

Keyboard API 可以用来监听键盘相关的事件。这里用一个例子来展示 Keyboard API 的用法，修改 `more.android.js` 代码如下：

```
01 import React, {Component} from 'react';
02 import {StyleSheet, Keyboard, View, Text, TextInput} from 'react-native';
03
04 import Screen from './Screen';
05
06 export default class more extends Component {
07   constructor(props) {
08     super(props);
09     this.state = {
10       keyboardText: '键盘收回'
11     }
12   }
13 }
```

```

14   componentWillMount() {
15     this.keyboardDidShowListener = Keyboard.addListener('keyboard
      DidShow', () => {
16       this.setState({keyboardText: '键盘弹出'});
17     });
18     this.keyboardDidHideListener = Keyboard.addListener('keyboard
      DidHide', () => {
19       this.setState({keyboardText: '键盘收回'});
20     });
21   }
22
23   render() {
24     return (
25       <View style={styles.container}>
26         <Text
27           style={styles.text}>{this.state.keyboardText}</Text>
28         <TextInput style={styles.textinput}/>
29       </View>
30     );
31   }
32
33   componentWillUnmount() {
34     this.keyboardDidShowListener.remove();
35     this.keyboardDidHideListener.remove();
36   }
37 }
38
39
40 const styles = StyleSheet.create({
41   // 这里省略了没有修改的代码
42   textinput: {
43     width: Screen.width,
44     height: 50,
45     backgroundColor: 'lightgray'
46   }
47 });

```

重新加载应用，可以看到在键盘弹出和收回时，会收到键盘状态变更的事件，效果如图 7.36 所示。



图 7.36 使用 Keyboard API 监听键盘事件

7.5.7 设备联网状态——NetInfo

NetInfo API 可以获知设备联网或离线的状态信息。这里用一个例子来展示 NetInfo API 的用法，修改 `more.android.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class more extends Component {
04   constructor(props) {
05     super(props);
06     this.state = {
07       connectionInfo: ''
08     }
09   }
10
11   componentDidMount() {
12     NetInfo.addEventListener('change', this._handleConnectionInfo
Change);
13     NetInfo.fetch().done((connectionInfo) => {
14       this.setState({connectionInfo});
15     });
16   }
17
18   render() {
19     return (
20       <View style={styles.container}>
21         <Text style={styles.text}>当前联网类型: {this.state.
connectionInfo}</Text>
22       </View>
23     );
24   }
25
26   componentWillUnmount() {
27     NetInfo.removeEventListener('change', this._handleConnection
InfoChange);
28   }
29
30   _handleConnectionInfoChange = (connectionInfo) => {
31     this.setState({connectionInfo});
32   }
33 }
34
35 // 这里省略了没有修改的代码

```

另外，和 Geolocation API 一样，NetInfo API 也需要请求权限，在 Android 工程的 `AndroidManifest.xml` 文件中添加如下配置：

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

重新编译和安装应用，打开“更多”页面，可以看到此时设备的联网类型如图 7.37 所示。




图 7.37 使用 NetInfo API 获取联网信息

7.5.8 权限设置——PermissionsAndroid

PermissionsAndroid API 可以使用 Android M（即 Android 7.0）开始提供的权限模型：有一些权限写在 AndroidManifest.xml 文件中就可以在安装时自动获得，但有一些“危险”的权限则需要弹出提示框供用户选择。PermissionsAndroid API 处理的就是后一种权限申请。

 **提示：**关于 Android 7.0 中权限模型的更多介绍，读者可以参考 Working with System Permissions (<https://developer.android.com/training/permissions/declaring.html#perm-needed>)。

 **注意：**为了使用 PermissionsAndroid API，需要使用 Android 7.0 及以上版本的 Android 设备。

这里以获取照相机权限为例，修改 more.android.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class more extends Component {
04   constructor(props) {
05     super(props);
06     this.state = {
07       permission: PermissionsAndroid.PERMISSIONS.CAMERA, '
08       hasPermission: 'Not Checked'
09     }
10   }
11
12   render() {
13     return (
14       <View style={styles.container}>
```

```
15         <Text style={styles.text}>权限状态: {this.state.hasPermission}</Text>
16         <Button title='申请摄像头权限'
17               onPress={this._requestCameraPermission}></Button>
18     </View>
19   );
20 }
21
22   _requestCameraPermission = async() => {
23     let result = await PermissionsAndroid.request(this.state.permission, {
24       title: '权限申请',
25       message: '申请摄像头权限'
26     });
27
28     this.setState({hasPermission: result});
29   };
30 }
31
32 // 这里省略了没有修改的代码
```

另外,和 Geolocation API 和 NetInfo API 一样,还需要在 Android 工程的 AndroidManifest.xml 文件中添加如下配置:

```
<uses-permission android:name="android.permission.CAMERA " />
```

重新编译和安装应用后,首次运行应用时还需要允许 permit drawing over other apps 权限,如图 7.38 所示。

获取 permit drawing over other apps 权限后,按“返回”键进入应用,接着,打开“更多”页面,单击“申请摄像头权限”按钮,效果如图 7.39 所示。

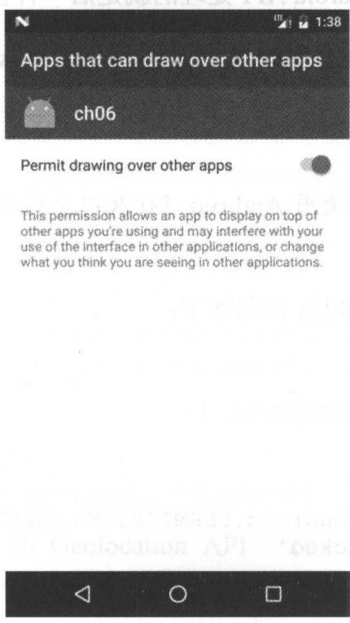


图 7.38 获取 permit drawing over other apps 权限

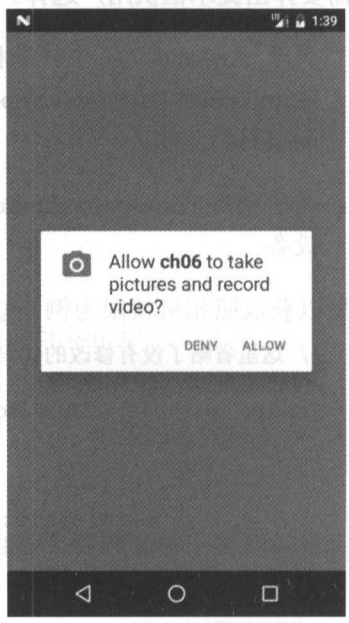


图 7.39 申请摄像头权限

 **提示:** 如果出现无法获取权限的问题, 请确保 Android 工程 android/app/build.gradle 文件中 targetSdkVersion 的配置不低于 Android 设备的版本, 例如, Android 设备版本为 24, 那么 targetSdkVersion 的配置也需要设置成 24。

7.5.9 悬浮提示框——ToastAndroid

ToastAndroid API 用于在 Android 设备上显示一个悬浮的提示信息。这里用一个例子来展示 ToastAndroid API 的用法, 修改 more.android.js 代码如下:

```
01 // 这里省略了没有修改的代码
02
03 export default class home extends Component {
04   // 这里省略了没有修改的代码
05
06   _onBackAndroid = () => {
07     // 最近 2 秒内按过“返回”键才退出应用
08     if (this.lastBackPressed && this.lastBackPressed + 2000 >=
09       Date.now()) {
10       return false;
11     }
12
13     this.lastBackPressed = Date.now();
14     ToastAndroid.show('再按一次退出应用', ToastAndroid.SHORT);
15     return true;
16   }
17   // 这里省略了没有修改的代码
18 }
```

重新加载应用, 按“返回”键后, 此时会弹出悬浮的提示框, 提示用户再按一次“返回”键退出应用, 效果如图 7.40 所示。



图 7.40 基于 ToastAndroid 的悬浮提示框

7.6 小 结

通过本章使用 React Native API 开发自定义组件的例子（包括获取分辨率以及自定义动画），不仅进一步加深了 React Native 组件的理解，还深入研究了 React Native API 的原理和机制。React Native 应用的详细架构如图 7.41 所示。

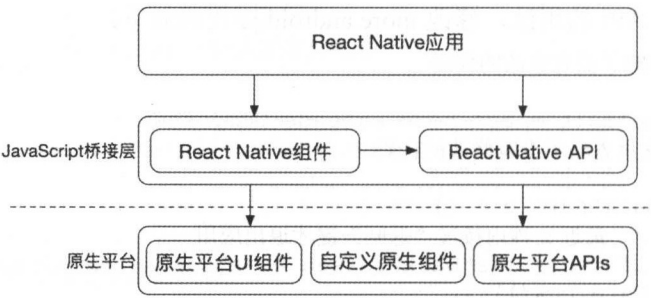


图 7.41 React Native 应用的整体架构

至此，基于 React Native 平台 JavaScript API 开发的所有知识都已经呈现在读者面前。但是在使用 React Native 进行实际开发时，难免会遇到以下这些情况：

- 需要使用 React Native 没有封装的原生功能。
- 复用已有的原生组件或原生的第三方组件。
- 多线程调用以及高性能要求的功能，例如加密、图像处理等。

为此，我们有时也需要编写原生代码以扩展 React Native 应用的功能，这也正是第 8 章要继续学习和讨论的内容。

第3篇

React Native 混合编程

» 第8章 React Native 与原生平台混合编程（1）

» 第9章 React Native 与原生平台混合编程（2）

» 第10章 电商 App 的复盘

目录

在 React Native 开发中，常常需要访问设备的相机、相册、设备信息、设备传感器、系统服务、系统设置、设备存储的资源，如相册中的照片等。

因此本章将以设备资源访问为例，来展示一下如何在 React Native 中实现应用功能的一般方式和流程。

(1) 仍然首先创建 React Native 项目，并安装 React Native 项目 ch07 中的代码。

(2) 将 ch06 项目中的如下目录复制到 ch07 文件夹中：

8.2 访问设备

• Animation.js

• App.js


在 7 章实现自己的 Platform 类，并实现原生平台的功能。这里将介绍如何获取更多的设备信息。

第8章 React Native 与原生平台混合编程（1）

通过前面章节的介绍可知：React Native 平台为开发者提供了强大的组件和 API 支持。但是在使用 React Native 进行实际开发时，难免会遇到以下这些情况：

- 需要使用 React Native 没有封装的原生功能。
- 复用已有的原生组件或原生的第三方组件。
- 多线程调用以及高性能要求的功能，如加密、图像处理等。

为此还需要编写原生代码以扩展 React Native 应用的功能，本章就带领读者冲破 React Native 平台的“小圈子”，走入广阔的原生开发世界。

提示：本章属于 React Native 开发进阶的内容，除了涉及 React Native 和 JavaScript 之外，还会有 iOS 与 Android 原生代码的编写。读者可以按照本书示例编写原生代码，或者参考 iOS/Android 开发的相关书籍和教程。

本章主要内容有：

- 访问设备。
- 访问相册。
- 了解 React Native 与原生平台的通信原理。
- 掌握 React Native 与原生页面的交互。

8.1 创建并移植项目

在 React Native 开发中，常常需要访问设备的相关资源。

- 设备信息，如设备名称、系统版本及系统语言等。
- 设备存储的资源，如相册中的图片等。

因此本节就以读取设备资源为例，来看一下编写原生代码扩展应用功能的一般方式和流程。

（1）仍然是先创建 React Native 项目并安装依赖包。

```
react-native init ch07          // 新建 React Native 项目 ch07
cd ch07
npm install
```

（2）将 ch06 项目中如下目录或文件都复制到 ch07 文件夹中：

- Animation.js;
- app.js;
- detail.js;
- home.android.js;

- home.ios.js;
- images;
- index.android.js;
- index.ios.js;
- main.android.js;
- main.ios.js;
- more.android.js;
- more.ios.js;
- Platform.js;
- Screen.js。

(3) 删除 more.android.js 文件, 并修改 more.ios.js 文件名为 more.js。

(4) 修改 index.ios.js 和 index.android.js 中的代码如下:

```
01 import React, {Component} from 'react';
02 import {AppRegistry} from 'react-native';
03 import app from './app';
04
05 AppRegistry.registerComponent('ch07', () => app); // 修改第一个参数为'ch07'
```

(5) 使用命令 react-native run-ios 或者 react-native run-android 运行 ch07 项目, 确保 ch06 项目的实现成功移植到 ch07 项目中, 效果如图 8.1 所示。




图 8.1 ch07 项目运行效果

8.2 访问设备

在第 7 章实现自己的 Platform API 中, 已经编写过简单的原生代码来实现获取设备名称的功能, 这里将介绍如何读取更多的设备信息。

(1) 新建一个文件 DeviceInfo.js, 用于提供设备信息的接口。添加 DeviceInfo.js 代码如下:

```
01 export default {
02   'systemName' : 'unknown',
03   'systemVersion' : 'unknown',
04   'defaultLanguage' : 'unknown',
05   'appVersion' : 'unknown'
06 }
```

提示: 这里的 DeviceInfo.js 文件中导出的是一个普通的模块, 而不是类似 home.js 文件使用 extends React.Component 声明的 React Native 组件。

(2) 在“更多”页面中显示设备相关信息。修改 more.js 代码如下:

```
01 import React, {Component} from 'react';
02 import {StyleSheet, View, Text} from 'react-native';
03
04 import DeviceInfo from './DeviceInfo';
05
06 export default class more extends Component {
07   render() {
08     return (
09       <View style={styles.container}>
10         <Text style={styles.text}>系统名称: {DeviceInfo.
11           systemName}</Text>
12         <Text style={styles.text}>系统版本: {DeviceInfo.
13           systemVersion}</Text>
14         <Text style={styles.text}>默认语言: {DeviceInfo.
15           defaultLanguage}</Text>
16         <Text style={styles.text}>应用版本: {DeviceInfo.
17           appVersion}</Text>
18       </View>
19     );
20   }
21 }
```

(3) 重新加载应用, 打开“更多”页面, 可以看到此时显示的设备信息为 unknown, 效果如图 8.2 所示。

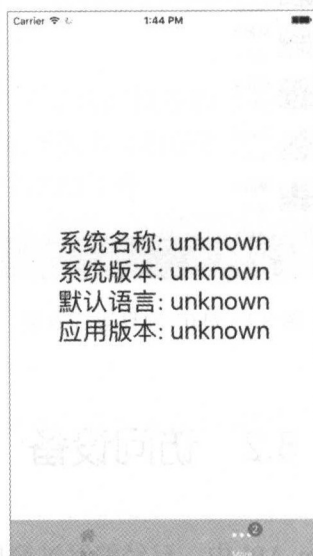


图 8.2 自定义 DeviceInfo API 初始效果

在准备好模块的定义和页面展示之后, 就可以专心开发与原生平台交互的接口了。

8.2.1 访问 iOS 设备

对于 iOS 平台, 使用 Xcode 打开 ios/ch08.xcodeproj 文件, 然后新建 DeviceInfo 类: 生成两个文件 DeviceInfo.m 和 DeviceInfo.h, 其中 DeviceInfo.m 是源文件, DeviceInfo.h 是头文件, 效果如图 8.3 所示。

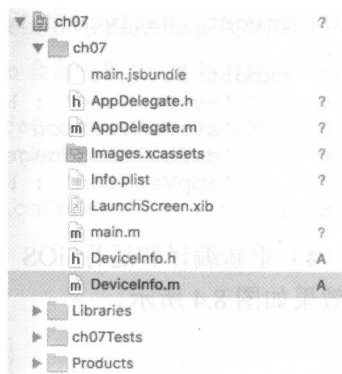


图 8.3 在 iOS 工程中新建 DeviceInfo 类

(1) 修改 DeviceInfo.h 代码如下:

```
01 #import <React/RCTBridgeModule.h>
02
03 @interface DeviceInfo : NSObject <RCTBridgeModule>
04
05 @end
```

其中, RCTBridgeModule 是 React Native 平台为开发者提供的与原生平台通信的桥梁接口。

(2) 实现原生代码的接口, 修改源文件 DeviceInfo.m 代码如下:

```
01 #import "Platform.h"
02 #import <UIKit/UIKit.h>
03
04 @implementation DeviceInfo
05
06 RCT_EXPORT_MODULE() // 导出此原生模块供 React Native 接口调用
07
08 - (NSDictionary *)constantsToExport {
09     UIDevice *currentDevice = [UIDevice currentDevice];
10
11     return @{
12         @"systemName": currentDevice.systemName, // 系统名称
13         @"systemVersion": currentDevice.systemVersion, // 系统版本
14         @"deviceLocale": self.deviceLocale, // 系统语言
15         @"appVersion": [[NSBundle mainBundle] objectForInfoDictionaryKey:@"CFBundleShortVersionString"], // 应用版本
16     }; // 返回键值对
17 }
18
19 - (NSString *)deviceLocale {
20     NSString *language = [[NSLocale preferredLanguages] objectAtIndex:0];
21     return language;
22 }
23
24 - (NSString *)deviceCountry {
25     NSString *country = [[NSLocale currentLocale] objectForKey:NSLocaleCountryCode];
26     return country;
27 }
28
29 @end
```

(3) React Native 平台中通过 NativeModules 调用原生平台实现的方式为 NativeModules.模块名称.接口名称。所以修改 DeviceInfo.js 代码如下:

```

01 import {NativeModules} from 'react-native';
02
03 export default {
04   'systemName' : NativeModules.DeviceInfo.systemName,
05   'systemVersion' : NativeModules.DeviceInfo.systemVersion,
06   'defaultLanguage' : NativeModules.DeviceInfo.deviceLocale,
07   'appVersion' : NativeModules.DeviceInfo.appVersion
08 }

```

(4) 重新编译和运行 iOS 应用, 查看“更多”页面, 此时显示就是有效的设备信息了, 效果如图 8.4 所示。



图 8.4 iOS 平台读取设备信息

8.2.2 访问 Android 设备

对于 Android 平台, 使用 Android Studio 打开 android 文件夹, 然后新建 DeviceInfoModule 类, 该类继承自 com.facebook.react.bridge.ReactContextBaseJavaModule, 效果如图 8.5 所示。

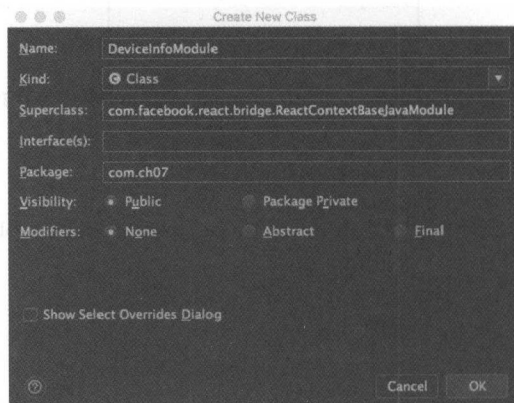


图 8.5 在 Android 工程中新建 DeviceInfoModule 类

(1) 新建 DeviceInfoModule 类成功后, 再修改 DeviceInfoModule.java 代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class DeviceInfoModule extends ReactContextBaseJavaModule {
04     private final ReactApplicationContext mReactContext;
05
06     public DeviceInfoModule(ReactApplicationContext reactContext) {
07         super(reactContext);
08         this.mReactContext = reactContext;
09     }
10
11     @Override
12     public String getName() {
13         return "DeviceInfo";          // 模块名称
14     }
15
16     @Override
17     public @Nullable
18     Map<String, Object> getConstants() {
19         HashMap<String, Object> constants = new HashMap<String, Object>();
20         constants.put("systemName", "Android");
21         constants.put("systemVersion", Build.VERSION.RELEASE);
22         constants.put("deviceLocale", this.getCurrentLanguage());
23         constants.put("appVersion", this.getAppVersion());
24         return constants;             // 返回键值对
25     }
26
27     // 这里省略了具体的方法实现
28 }

```

(2) 其中 getCurrentLanguage() 和 getAppVersion() 方法的具体实现如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class DeviceInfoModule extends ReactContextBaseJavaModule {
04     // 这里省略了上述已经描述的代码
05
06     private String getCurrentLanguage() {
07         Locale current = getReactApplicationContext().getResources().
08             getConfiguration().locale;
09         if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
10             return current.toLanguageTag();
11         } else {
12             StringBuilder builder = new StringBuilder();
13             builder.append(current.getLanguage());
14             if (current.getCountry() != null) {
15                 builder.append("-");
16                 builder.append(current.getCountry());
17             }
18             return builder.toString();
19         }
20     }
21
22     private String getAppVersion() {
23         String appVersion = "not available";
24         try {
25             PackageManager packageManager = this.mReactContext.get
26                 PackageManager();
27             String packageName = this.mReactContext.getPackageName();
28             PackageInfo info = packageManager.getPackageInfo(packageName, 0);

```

```

27         constants.put("appVersion", info.versionName);
28     } catch (PackageManager.NameNotFoundException e) {
29         e.printStackTrace();
30     }
31     return appVersion;
32 }
33 }

```

(3) 与 iOS 平台导出模块类似, Android 平台也需要将 DeviceInfo 模块导出供 React Native 接口调用。

这里再新建 DeviceInfo.java 文件, 并添加代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class DeviceInfo implements ReactPackage {
04     @Override
05     public List<NativeModule> createNativeModules(
06         ReactApplicationContext reactContext) {
07         List<NativeModule> modules = new ArrayList<>();
08         modules.add(new DeviceInfoModule(reactContext));
09         return modules;
10     }
11
12     @Override
13     public List<Class<? extends JavaScriptModule>> createJSModules() {
14         return Collections.emptyList();
15     }
16
17     @Override
18     public List<ViewManager> createViewManagers(
19         ReactApplicationContext reactContext) {
20         return Collections.emptyList();
21     }
22 }

```

(4) 还需要在 MainApplication.java 文件中注册, 才算正式完成导出 DeviceInfo 模块的操作。修改 MainApplication.java 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 public class MainApplication extends Application implements React
    Application {
04
05     private final ReactNativeHost mReactNativeHost = new ReactNative
        Host(this) {
06         // 这里省略了没有修改的代码
07
08         @Override
09         protected List<ReactPackage> getPackages() {
10             return Arrays.<ReactPackage>asList(
11                 new MainReactPackage(), new DeviceInfo()
12                                     // 导出 DeviceInfo 模块
13                                     );
14         }
15     };
16
17     // 这里省略了没有修改的代码
18 }

```

(5) 重新编译和运行 Android 应用, 查看“更多”页面, 此时显示的就是有效的设备信息了, 效果如图 8.6 所示。

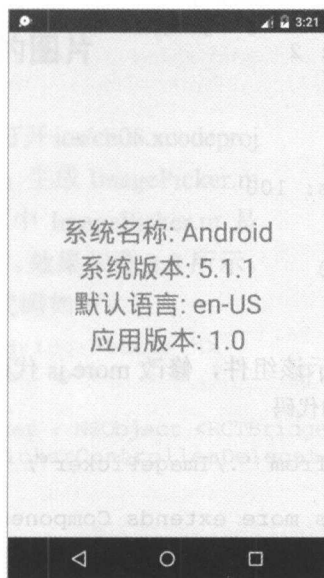


图 8.6 Android 平台读取设备信息

8.3 访问相册

8.2 节为 React Native 应用扩展了读取设备信息的功能, 由于已经有了 7.4 节自己实现 Platform 的经验, 想必读者还是觉得很轻松的。本节将实现一个更复杂的原生接口: 读取相册中的图片。

(1) 创建组件, 新建 ImagePicker.js 文件, 添加代码如下:

```
01 import React, {Component} from 'react';
02 import {StyleSheet, Text, View, TouchableOpacity} from 'react-native';
03
04 export default class ImagePicker extends Component {
05   render() {
06     return (
07       <View style={styles.container}>
08         <TouchableOpacity>
09           <View style={[styles.avatarContainer, styles.avatar]}>
10             <Text style={styles.text}>选择图片</Text>
11           </View>
12         </TouchableOpacity>
13       </View>
14     );
15   }
16 }
17
18 const styles = StyleSheet.create({
19   container: {
20     alignSelf: 'center',
21     flexDirection: 'row'
```



```

22     },
23     avatarContainer: {
24       justifyContent: 'center',
25       alignItems: 'center',
26       borderColor: 'lightgray',
27       borderWidth: 2
28     },
29     avatar: {
30       width: 200,
31       height: 200,
32       borderRadius: 100
33     },
34     text: {
35       fontSize: 30
36     }
37   });

```

(2) 在“更多”页面中显示该组件，修改 `more.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 import ImagePicker from './ImagePicker';
04
05 export default class more extends Component {
06   render() {
07     return (
08       <View style={styles.container}>
09         <ImagePicker/>
10       </View>
11     );
12   }
13 }
14
15 // 这里省略了没有修改的代码

```

(3) 重新加载应用，打开“更多”页面，可以看到封装的 `ImagePicker` 组件效果如图 8.7 所示。



图 8.7 自定义 `ImagePicker` 组件

在准备好模块的定义和页面展示之后, 就可以专心开发与原生平台交互的接口了。

8.3.1 读取 iOS 相册中的图片

对于 iOS 平台, 使用 Xcode 打开 ios/ch08.xcodeproj 文件, 然后新建 ImagePicker 类: 生成 ImagePicker.m 和 ImagePicker.h 两个文件, 其中 ImagePicker.m 是源文件, ImagePicker.h 是头文件, 效果如图 8.8 所示。

(1) 修改 ImagePicker.h 代码如下:

```
01 #import <React/RCTBridgeModule.h>
02 #import <UIKit/UIKit.h>
03
04 @interface ImagePicker : NSObject <RCTBridgeModule, UINavigationController
  Delegate, UIImagePickerControllerDelegate>
05
06 @end
```

图 8.8 在 iOS 工程中新建 ImagePicker 类

(2) 实现原生代码的接口, 在 iOS 开发中源文件主要有两部分组成:

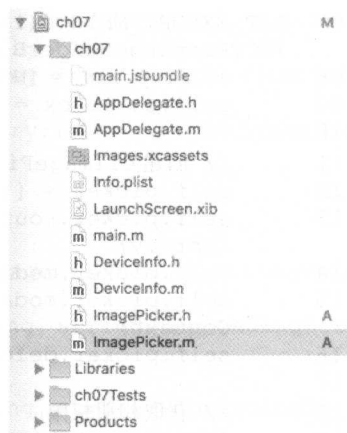
- 头文件和接口定义。
- 类接口的详细实现。

首先导入头文件和定义接口, 修改 ImagePicker.m 代码如下:

```
01 #import "ImagePicker.h"
02 #import <React/RCTConvert.h>
03 #import <AssetsLibrary/AssetsLibrary.h>
04 #import <AVFoundation/AVFoundation.h>
05 #import <Photos/Photos.h>
06
07 @import MobileCoreServices;
08
09 @interface ImagePicker ()
10
11 @property (nonatomic, retain) NSMutableDictionary *options, *response;
12 @property (nonatomic, strong) RCTResponseSenderBlock callback;
13 @property (nonatomic, strong) UIImagePickerController *picker;
14
15 @end
16
17 @implementation ImagePicker
18
19 // 这里省略了具体的方法实现
20
21 @end
```

然后实现 launchImagePicker 接口, 修改 ImagePicker.m 代码如下:

```
01 // 这里省略了上述已经描述的代码
02
03 @implementation ImagePicker
04
05 RCT_EXPORT_MODULE();
06
```



```

07  RCT_EXPORT_METHOD(launchImagePicker:(NSDictionary *)options callback:
    (RCTResponseSenderBlock)callback) {
08      self.options = [NSMutableDictionary dictionaryWithDictionary: options];
09      self.callback = callback;
10
11      // 创建 UIImagePickerController
12      self.picker = [[UIImagePickerController alloc] init];
13      self.picker.sourceType = UIImagePickerControllerSourceTypePhoto
        Library;
14      self.picker.mediaTypes = @[(NSString *)kUTTypeImage];
15      self.picker.modalPresentationStyle = UIModalPresentationCurrent
        Context;
16      self.picker.delegate = self;
17
18      // 获取相册权限
19      [self checkPhotosPermissions:^(BOOL granted) {
20          if (!granted) {
21              self.callback(@[@"error": @"Photo library permissions
                not granted"]);
22              return;
23          }
24
25          // 打开 UIImagePickerController
26          dispatch_async(dispatch_get_main_queue(), ^{
27              UIViewController *root = [[[UIApplication]
                delegate] window] rootViewController];
28              while (root.presentedViewController != nil) {
29                  root = root.presentedViewController;
30              }
31              [root presentViewController:self.picker animated:YES completion:
                nil];
32          });
33      }];
34  }
35
36  // 这里省略了具体的辅助接口实现
37
38  @end

```

(3) 在用户进行选择图片操作之后, 原生代码需要对选择的图片进行处理, 然后将图片返回给 React Native 平台, 该实现涉及的辅助接口代码如下:

```

01  @implementation ImagePicker
02
03  // 这里省略了上述已经描述的代码
04
05  #pragma mark - UIImagePickerControllerDelegate
06
07  - (void)imagePickerController:(UIImagePickerController *)picker did
    FinishPickingMediaWithInfo:(NSDictionary<NSString *,id> *)info {
        // 选择某一图片
08      UIImage *image = [info objectForKey:UIImagePickerControllerOriginal
        Image];
09
10      // 图片缩放
11      float maxWidth = image.size.width;
12      float maxHeight = image.size.height;
13      if ([self.options valueForKey:@"maxWidth"]) {
14          maxWidth = [[self.options valueForKey:@"maxWidth"] floatValue];

```

```

15     }
16     if ([self.options valueForKey:@"maxHeight"]) {
17         maxHeight = [[self.options valueForKey:@"maxHeight"] floatValue];
18     }
19     image = [self downscaleImageIfNecessary:image maxWidth:maxWidth
20             maxHeight:maxHeight];
21     // 设置图片路径
22     NSString *fileName;
23     if ([[[self.options objectForKey:@"imageFileType"] stringValue]
24         isEqualToString:@"png"]) {
25         fileName = [[NSUUID UUID] UUIDString] stringByAppendingString:
26             @".png";
27     } else {
28         fileName = [[NSUUID UUID] UUIDString] stringByAppendingString:
29             @".jpg";
30     }
31     NSString *path = [[NSTemporaryDirectory()]stringByStandardizing
32         Path] stringByAppendingString:fileName];
33     NSData *data = UIImageJPEGRepresentation(image, [[self.options
34         valueForKey:@"quality"] floatValue]);
35     [data writeToFile:path atomically:YES];
36
37     self.response = [[NSMutableDictionary alloc] init];
38
39     // 设置图片 uri
40     NSURL *fileURL = [NSURL fileURLWithPath:path];
41     NSString *filePath = [fileURL absoluteString];
42     [self.response setObject:filePath forKey:@"uri"];
43
44     // 设置图片大小
45     NSNumber *fileSizeValue = nil;
46     NSError *fileSizeError = nil;
47     [fileURL getResourceValue:&fileSizeValue forKey:NSURLFileSizeKey
48         error:&fileSizeError];
49     if (fileSizeValue){
50         [self.response setObject:fileSizeValue forKey:@"fileSize"];
51     }
52
53     // 回调函数
54     self.callback(@[self.response]);
55
56     // 关闭 UIImagePickerController
57     [picker dismissViewControllerAnimated:YES completion:nil];
58 }
59
60 - (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker {
61     // 取消图片选择
62
63     // 回调函数
64     self.callback(@[ @{@"didCancel": @YES} ]);
65
66     // 关闭 UIImagePickerController
67     [picker dismissViewControllerAnimated:YES completion:nil];
68 }
69
70 // 这里省略了具体的辅助接口实现
71
72 @end

```

上述代码中用到的 `checkPhotosPermissions()` 和 `downscaleImageIfNecessary()` 具体实现如下:

```

01  @implementation UIImagePickerController
02
03  // 这里省略了上述已经描述的代码
04
05  #pragma mark - Helpers
06
07  - (void)checkPhotosPermissions:(void(^)(BOOL granted))callback {
08      PHAuthorizationStatus status = [PHPhotoLibrary authorization
09      Status];
10      if (status == PHAuthorizationStatusAuthorized) {
11          callback(YES);
12          return;
13      } else if (status == PHAuthorizationStatusNotDetermined) {
14          [PHPhotoLibrary requestAuthorization:^(PHAuthorizationStatus
15          status) {
16              if (status == PHAuthorizationStatusAuthorized) {
17                  callback(YES);
18                  return;
19              } else {
20                  callback(NO);
21                  return;
22              }
23          }];
24      } else {
25          callback(NO);
26      }
27  }
28
29  - (UIImage*)downscaleImageIfNecessary:(UIImage*)image maxWidth:(float)
30  maxWidth maxHeight:(float)maxHeight {
31      UIImage* newImage = image;
32      if (image.size.width <= maxWidth && image.size.height <= maxHeight) {
33          return newImage;
34      }
35
36      // 计算大小
37      CGSize scaledSize = CGSizeMake(image.size.width, image.size.
38      height);
39      if (maxWidth < scaledSize.width) {
40          scaledSize = CGSizeMake(maxWidth, (maxWidth / scaledSize.width)
41          * scaledSize.height);
42      }
43      if (maxHeight < scaledSize.height) {
44          scaledSize = CGSizeMake((maxHeight / scaledSize.height) *
45          scaledSize.width, maxHeight);
46      }
47      scaledSize.width = (int)scaledSize.width;
48      scaledSize.height = (int)scaledSize.height;
49
50      // 缩放图片
51      UIGraphicsBeginImageContext(scaledSize);
52      [image drawInRect:CGRectMake(0, 0, scaledSize.width, scaledSize.
53      height)];
54      newImage = UIGraphicsGetImageFromCurrentImageContext();
55      if (newImage == nil) {
56          NSLog(@"could not scale image");
57      }
58  }

```

```

51     UIGraphicsEndImageContext();
52
53     return newImage;
54 }
55
56 @end

```

(4) 完成了原生代码之后, 还需要完善 ImagePicker 组件: 调用 iOS 原生代码的接口并使用返回的图片, 修改 ImagePicker.js 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 export default class ImagePicker extends Component {
04     constructor(props) {
05         super(props);
06         this.state = {
07             avatarSource: null
08         };
09     }
10
11     render() {
12         return (
13             <View style={styles.container}>
14                 <TouchableOpacity onPress={this._selectPhotoTapped}>
15                     <View style={[styles.avatarContainer, styles.avatar]}>
16                         {this.state.avatarSource === null
17                             ? <Text style={styles.text}>选择图片</Text>
18                             : <Image style={styles.avatar}
19                                 source={this.state.avatarSource}/>
20                         }
21                     </View>
22                 </TouchableOpacity>
23             </View>
24         );
25     }
26
27     _selectPhotoTapped = () => {
28         const options = {
29             quality: 1.0,
30             maxWidth: 500,
31             maxHeight: 500
32         };
33
34         NativeModules.ImagePicker.launchImagePicker(options, (response)
35             => {
36             if (response.didCancel) {
37                 console.log('取消选择图片');
38             } else if (response.error) {
39                 console.log('选择图片错误: ', response.error);
40             } else {
41                 let source = {
42                     uri: response.uri.replace('file://', '')
43                 };
44                 this.setState({avatarSource: source});
45             }
46         });
47     }
48
49 // 这里省略了没有修改的代码

```

重新编译和运行 iOS 应用，打开“更多”页面，单击“选择图片”按钮，然后选择某张图片，效果如图 8.9 和图 8.10 所示。



图 8.9 iOS 平台读取图片资源



图 8.10 具体图片

8.3.2 读取 Android 相册中的图片

对于 Android 平台，使用 Android Studio 打开 android 文件夹，然后新建 ImagePicker Module 类，该类继承自 com.facebook.react.bridge.ReactContextBaseJavaModule，效果如图 8.11 所示。

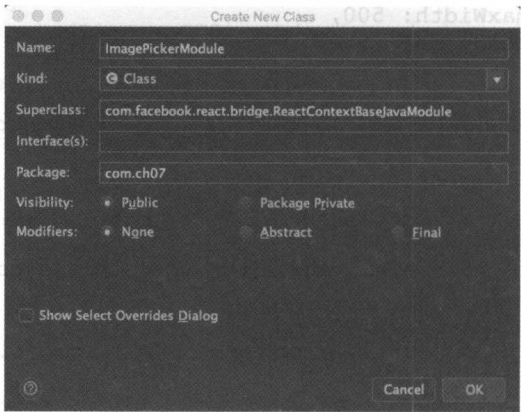


图 8.11 在 Android 工程中新建 ImagePickerModule 类

(1) 实现 launchImagePicker 接口，修改 ImagePickerModule.java 代码如下：

```
01 // 这里省略了导入文件的代码，因为 Android Studio 会自动导入所依赖的模块
02
```



```

03 public class ImagePickerModule extends ReactContextBaseJavaModule
    implements ActivityEventListener {
04     static final int REQUEST_LAUNCH_IMAGE_LIBRARY = 13002;
05     private final ReactApplicationContext mReactContext;
06     private Callback mCallback;
07     private WritableMap mResponse;
08
09     public ImagePickerModule(ReactApplicationContext reactContext) {
10         super(reactContext);
11         mReactContext = reactContext;
12         reactContext.addActivityEventListener(this);
13     }
14
15     @Override
16     public String getName() {
17         return "ImagePicker";           // 模块名称
18     }
19
20     @ReactMethod
21     public void launchImagePicker(final ReadableMap options, final
        Callback callback) {
22         mCallback = callback;
23         mResponse = Arguments.createMap();
24
25         Intent libraryIntent = new Intent(Intent.ACTION_PICK,
26             MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
27         try {
28             // 打开图库
29             int requestCode = REQUEST_LAUNCH_IMAGE_LIBRARY;
30             Activity currentActivity = getCurrentActivity();
31             currentActivity.startActivityForResult(libraryIntent,
32                 requestCode);
33         } catch (ActivityNotFoundException e) {
34             e.printStackTrace();
35             if (mCallback != null) {
36                 mResponse.putString("error", "Cannot launch photo
37                     library");
38                 mCallback.invoke(mResponse);
39                 mCallback = null;
40             }
41         }
42         // 这里省略了具体的辅助接口实现
43     }

```

(2) 在用户进行选择图片操作之后, 原生代码需要对选择的图片进行处理, 然后将图片返回给 React Native 平台, 该实现涉及的辅助接口代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class ImagePickerModule extends ReactContextBaseJavaModule
    implements ActivityEventListener {
04     // 这里省略了上述已经描述的代码
05
06     public void onActivityResult(Activity activity, int requestCode,
        int resultCode, Intent data) {
07         mResponse = Arguments.createMap();
08
09         // 用户取消

```

```

10         if (resultCode != Activity.RESULT_OK) {
11             mResponse.putBoolean("didCancel", true);
12             mCallback.invoke(mResponse);
13             mCallback = null;
14             return;
15         }
16
17         Uri uri = data.getData();
18         String realPath = getRealPathFromURI(uri);
19         if (!TextUtils.isEmpty(realPath)) {
20             // 解码图片
21             Options options = new Options();
22             options.inJustDecodeBounds = true;
23             BitmapFactory.decodeFile(realPath, options);
24
25             // 回调函数
26             mResponse.putString("uri", uri.toString());
27             mResponse.putString("path", realPath);
28             mCallback.invoke(mResponse);
29             mCallback = null;
30         } else {
31             if (mCallback != null) {
32                 mResponse.putString("error", "Cannot launch photo
33                     library");
34                 mCallback.invoke(mResponse);
35                 mCallback = null;
36             }
37         }
38
39         public void onNewIntent(Intent intent) {
40         }
41
42         private String getRealPathFromURI(Uri uri) {
43             String result;
44             String[] projection = {MediaStore.Images.Media.DATA};
45             if (uri == null || TextUtils.isEmpty(uri.toString())) {
46                 return "";
47             }
48             Cursor cursor = mReactContext.getContentResolver().query(uri,
49                 projection, null, null, null);
50             if (cursor == null) {
51                 result = uri.getPath();
52             } else {
53                 cursor.moveToFirst();
54                 int idx = cursor.getColumnIndexOrThrow(MediaStore.Images.
55                     Media.DATA);
56                 result = cursor.getString(idx);
57                 cursor.close();
58             }
59             return result;
60         }

```

(3) 和访问设备的例子类似, Android 项目还需要将 ImagePicker 模块导出供 React Native 接口调用。这里再新建 ImagePicker.java 文件, 并添加代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class ImagePicker implements ReactPackage {
04     @Override

```

```

05     public List<NativeModule> createNativeModules(
06         ReactApplicationContext reactContext) {
07         List<NativeModule> modules = new ArrayList<>();
08         modules.add(new ImagePickerModule(reactContext));
09         return modules;
10     }
11
12     @Override
13     public List<Class<? extends JavaScriptModule>> createJSMModules() {
14         return Collections.emptyList();
15     }
16
17     @Override
18     public List<ViewManager> createViewManagers(
19         ReactApplicationContext reactContext) {
20         return Collections.emptyList();
21     }
22 }

```

(4) 还需要在 `MainApplication.java` 文件中注册, 才算正式完成导出 `ImagePicker` 模块的操作。修改 `MainApplication.java` 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 public class MainApplication extends Application implements React
04     Application {
05     private final ReactNativeHost mReactNativeHost = new ReactNative
06         Host(this) {
07         // 这里省略了没有修改的代码
08
09         @Override
10         protected List<ReactPackage> getPackages() {
11             return Arrays.<ReactPackage>asList(
12                 new MainReactPackage(), new DeviceInfo(), new Image
13                 Picker() // 导出 ImagePicker 模块
14             );
15         }
16     };
17 // 这里省略了没有修改的代码
18 }

```

(5) 完成了原生代码之后, 还需要完善 `ImagePicker` 组件: 调用 `Android` 原生代码的接口并使用返回的图片, 修改 `ImagePicker.js` 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 export default class ImagePicker extends Component {
04     // 这里省略了没有修改的代码
05
06     _selectPhotoTapped = () => {
07         const options = {
08             quality: 1.0,
09             maxWidth: 500,
10             maxHeight: 500
11         };
12
13         NativeModules.ImagePicker.launchImagePicker(options, (response)
14             => {

```

```

14         if (response.didCancel) {
15             console.log('取消选择图片');
16         } else if (response.error) {
17             console.log('选择图片错误: ', response.error);
18         } else {
19             let source;
20             if (Platform.OS === 'ios') {
21                 source = {
22                     uri: response.uri.replace('file://', '')
23                 };
24             } else if (Platform.OS === 'android') {
25                 source = {
26                     uri: response.uri
27                 };
28             }
29             this.setState({avatarSource: source});
30         }
31     });
32 }
33 }
34
35 // 这里省略了没有修改的代码

```

(6) 重新编译和运行 Android 应用，打开“更多”页面，单击“选择图片”按钮，然后选择某张图片，效果如图 8.12 和图 8.13 所示。

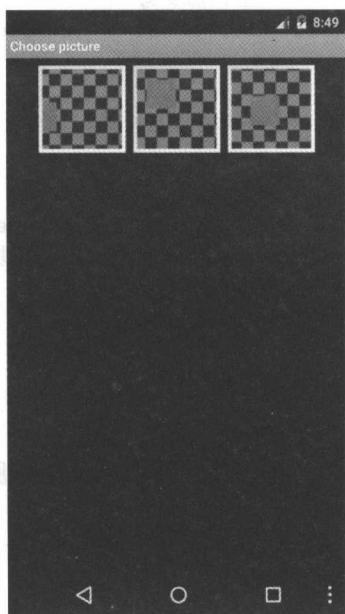


图 8.12 Android 平台读取图片资源

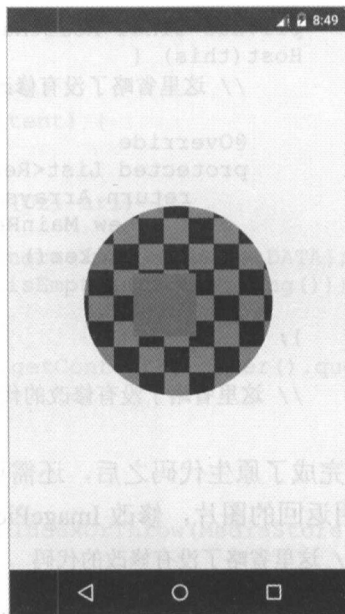


图 8.13 具体图片

8.4 React Native 与原生平台的通信原理

通过访问设备的例子可以看出 React Native 平台与原生平台的通信原理，如图 8.14 所示。

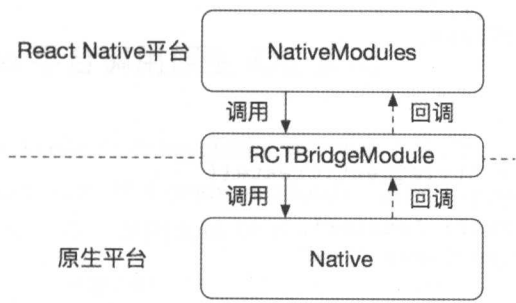


图 8.14 React Native 与原生平台通信原理

其中：

- React Native 平台调用原生平台基于 NativeModules，调用的方法是 NativeModules. 模块名称.接口名称。
- 原生平台返回数据到 React Native 平台基于回调，回调的原型定义是 RCTResponse SenderBlock（iOS 平台）和 com.facebook.react.bridge.Callback（Android 平台）。

当 JavaScript 接口调用原生代码时，React Native 与原生平台的数据类型对应关系如表 8.1 所示。

表 8.1 React Native与原生平台的数据类型对应关系

| React Native | iOS平台 | Android平台 |
|--------------|---|------------------------------------|
| string | NSString | String |
| number | NSInteger、float、double、CGFloat、NSNumber | Integer、Double、Float |
| boolean | BOOL、NSNumber | Bool |
| array | NSArray | List |
| map | NSDictionary | Map |
| function | RCTResponseSenderBlock | com.facebook.react.bridge.Callback |

8.5 React Native 平台调用原生页面

在了解了 React Native 与原生平台通信的原理之后，下面通过两者交互的实例看一下详细的通信过程。

(1)创建一个用于实现通信功能的 Communication 组件，新建 Communication.js 文件，添加代码如下：

```
01 import React, {Component} from 'react';
02 import {StyleSheet, View, Button, Alert} from 'react-native';
03
04 export default class Communication extends Component {
05   render() {
06     return (
07       <View style={styles.container}>
08         <Button title='调用原生组件' onPress={() => {
09           Alert.alert('调用原生组件', null, null);
```

```

10         }}/>
11     </View>
12   );
13 }
14 }
15
16 const styles = StyleSheet.create({
17   container: {
18     alignSelf: 'center',
19     flexDirection: 'row'
20   }
21 });

```

(2) 在“更多”页面中显示该组件，修改 `more.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 import Communication from './Communication';
04
05 export default class more extends Component {
06   render() {
07     return (
08       <View style={styles.container}>
09         <Communication/>
10       </View>
11     );
12   }
13 }
14
15 // 这里省略了没有修改的代码

```

(3) 重新加载应用，打开“更多”页面，可以看到封装的 `Communication` 组件效果如图 8.15 所示。



图 8.15 自定义 `Communication` 组件

在准备好组件和页面展示之后，就可以专心开发与原生平台交互的接口了。

8.5.1 React Native 平台调用原生 iOS 页面

对于 iOS 平台, 使用 Xcode 打开 ios/ch08.xcodeproj 文件, 然后新建 Communication 类: 生成两个文件 Communication.m 和 Communication.h, 其中 Communication.m 是源文件, Communication.h 是头文件, 效果如图 8.16 所示。

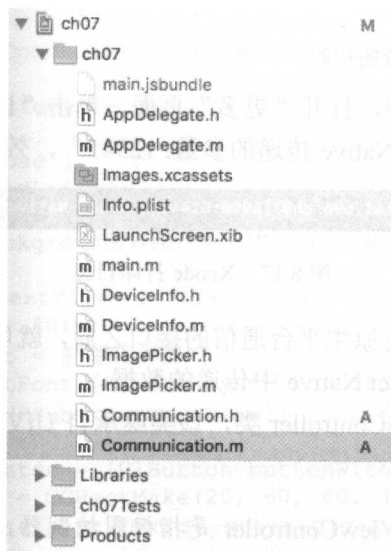


图 8.16 在 iOS 工程中新建 Communication 类

(1) 修改 Communication.h 代码如下:

```
01 #import <React/RCTBridgeModule.h>
02
03 @interface Communication : NSObject<RCTBridgeModule>
04
05 @end
```

(2) 实现原生代码的接口, 修改源文件 Communication.m 代码如下:

```
01 #import "Communication.h"
02
03 @implementation Communication
04
05 RCT_EXPORT_MODULE();
06
07 RCT_EXPORT_METHOD(presentViewControllerFromReactNative:(NSString
08 *)params) {
09     NSLog(@"React Native 传递的参数: %@", params);    // 打印传递的参数
10 }
11 @end
```

(3) 在 React Native 代码中调用该原生接口, 修改 Communication.js 代码如下:

```
01 // 这里省略了没有修改的代码
02
03 export default class Communication extends Component {
04     render() {
```

```

05         return (
06             <View style={styles.container}>
07                 <Button title='调用原生组件' onPress={() => {
08                     NativeModules.Communication.presentViewController
09                         FromReactNative('12345');
10                 }}/>
11             </View>
12         );
13     }
14
15     // 这里省略了没有修改的代码

```

重新编译和运行 iOS 应用，打开“更多”页面，单击“调用原生组件”按钮，可以看到 Xcode 中打印日志“React Native 传递的参数: 12345”，效果如图 8.17 所示。

2017-02-13 10:21:41.696 ch07[5990:514486] React Native传递的参数: 12345

图 8.17 Xcode 打印日志

(4) 连通 React Native 与原生平台通信的接口之后，就可以继续完善原生接口了：打开一个原生界面并显示从 React Native 中传递的数据。

新建 CommunicationViewController 类，该类继承自 UIViewController，效果如图 8.18 所示。

 **小知识：** iOS 开发中的 UIViewController 是指视图控制器，简单来说，可以理解每一个 UIViewController 以及继承自 UIViewController 的类都是一个界面。

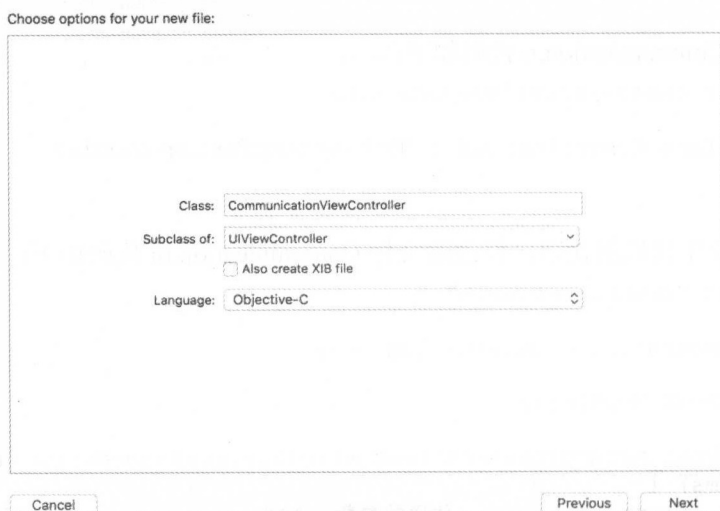


图 8.18 在 iOS 工程中新建 CommunicationViewController 类

(5) 修改头文件 CommunicationViewController.h 代码如下：

```

01 #import <UIKit/UIKit.h>
02
03 @interface CommunicationViewController : UIViewController
04

```



```

05 @property (nonatomic, copy) NSString *params;
06
07 @end

```

(6) 完善原生界面显示的内容, 修改源文件 CommunicationViewController.m 代码如下:

```

01 #import "CommunicationViewController.h"
02
03 @interface CommunicationViewController ()
04
05 @end
06
07 @implementation CommunicationViewController
08
09 #pragma mark - Lifecycle
10
11 - (void)viewDidLoad {
12     [super viewDidLoad];
13
14     self.view.backgroundColor = [UIColor whiteColor];
15
16     UITextView *textView = [[UITextView alloc] initWithFrame:CGRectMake(
17         20, 20, 200, 40)];
18     textView.text = @"原生界面";
19     [textView setFont:[UIFont systemFontOfSize:20]];
20     [self.view addSubview:textView];    // 显示“原生界面”提示信息
21
22     UIButton *button = [UIButton buttonWithType:UIButtonTypeSystem];
23     button.frame = CGRectMake(20, 60, 60, 40);
24     [button setTitle:@"退出" forState:UIControlStateNormal];
25     [button addTarget:self action:@selector(buttonClicked:) for
26        ControlEvents:UIControlEventTouchUpInside];
27     [self.view addSubview:button];    // 退出原生界面的按钮
28 }
29
30 - (void)viewDidAppear:(BOOL)animated {
31     [super viewDidAppear:animated];
32
33     UIAlertController *alertController = [UIAlertController alert
34         ControllerWithTitle:@"从 React Native 传来的数据是:" message:self.
35         params preferredStyle:UIAlertControllerStyleAlert];
36     UIAlertAction *cancelAction = [UIAlertAction actionWithTitle:@"取
37         消" style:UIAlertActionStyleCancel handler:nil];
38     [alertController addAction:cancelAction];
39     [self presentViewController:alertController animated:YES completion:
40         nil];    // 显示 React Native 传递的参数
41 }
42
43 #pragma mark - IBActions
44
45 - (void)buttonClicked:(UIButton *)button {    // 按钮响应事件
46     [self dismissViewControllerAnimated:YES completion:nil];
47 }
48
49 @end

```

(7) 修改用于通信的 Communication 中的逻辑, 修改 Communication.m 代码如下:

```

01 #import "Communication.h"
02 #import "CommunicationViewController.h"
03 #import "AppDelegate.h"

```

```

04
05 @implementation Communication
06
07 RCT_EXPORT_MODULE();
08
09 - (dispatch_queue_t)methodQueue {
10     return dispatch_get_main_queue(); // 因为是显示界面，所以让原生接口运
                                           行在主线程
11 }
12
13 RCT_EXPORT_METHOD(presentViewControllerFromReactNative:(NSString
*)params) {
14     CommunicationViewController *communicationViewController =
        [[CommunicationViewController alloc] init];
15     communicationViewController.params = params;
16
17     AppDelegate *app = (AppDelegate *)[[UIApplication sharedApplication]
        delegate];
18     UIViewController *rootViewController = (UIViewController *)[[app
        window] rootViewController];
19
20     [rootViewController presentViewController:communicationViewController
        animated:YES completion:nil];
21 }
22
23 @end

```

(8) 重新编译和运行 iOS 应用，打开“更多”页面，单击“调用原生组件”按钮，此时就打开了一个原生界面并显示 React Native 传递的参数，效果如图 8.19 所示。

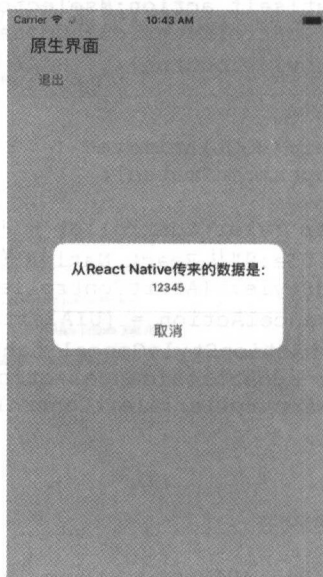


图 8.19 React Native 平台调用 iOS 原生页面

8.5.2 React Native 平台调用原生 Android 页面

对于 Android 平台，使用 Android Studio 打开 android 文件夹，然后新建 Communication Module 类并实现 `startActivityFromReactNative` 接口。

(1) 修改 CommunicationModule.java 代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class CommunicationModule extends ReactContextBaseJavaModule {
04
05     private final ReactApplicationContext mReactContext;
06
07     public CommunicationModule(ReactApplicationContext reactContext) {
08         super(reactContext);
09         this.mReactContext = reactContext;
10     }
11
12     @Override
13     public String getName() {
14         return "Communication"; // 模块名称
15     }
16
17     @ReactMethod
18     public void startActivityFromReactNative(String activityName,
19         String params) {
20         try {
21             Activity currentActivity = getCurrentActivity();
22             if (currentActivity != null) {
23                 Class toActivity = Class.forName(activityName);
24                 Intent intent = new Intent(currentActivity, toActivity);
25                 intent.putExtra("params", params);
26                 currentActivity.startActivity(intent);
27             }
28         } catch (Exception e) {
29             throw new JSApplicationIllegalArgumentException("打开
30             Activity 失败: " + e.getMessage());
31         }
32     }

```

(2) Android 项目还需要将 Communication 模块导出供 React Native 接口调用。新建 Communication.java 文件, 添加代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class Communication implements ReactPackage {
04     @Override
05     public List<NativeModule> createNativeModules(
06         ReactApplicationContext reactContext) {
07         List<NativeModule> modules = new ArrayList<>();
08         modules.add(new CommunicationModule(reactContext));
09         return modules;
10     }
11
12     @Override
13     public List<Class<? extends JavaScriptModule>> createJSModules() {
14         return Collections.emptyList();
15     }
16
17     @Override
18     public List<ViewManager> createViewManagers(
19         ReactApplicationContext reactContext) {
20         return Collections.emptyList();
21     }
22 }

```

(3) 在 MainApplication.java 文件中注册 Communication 模块, 修改 MainApplication.java 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 public class MainApplication extends Application implements React
    Application {
04
05     private final ReactNativeHost mReactNativeHost = new React
        NativeHost(this) {
06         // 这里省略了没有修改的代码
07
08         @Override
09         protected List<ReactPackage> getPackages() {
10             return Arrays.<ReactPackage>asList(
11                 new MainReactPackage(), new DeviceInfo(), new Image
                    Picker(), new Communication() // 导出Communication模块
                );
12         }
13     };
14 };
15
16 // 这里省略了没有修改的代码
17 }

```

(4) 完成了原生代码之后, 还需要修改 Communication 组件的逻辑: 针对不同原生平台调用相应的接口, 修改 Communication.js 代码如下:

```

01 // 这里省略了没有修改的代码
02
03 export default class Communication extends Component {
04     render() {
05         return (
06             <View style={styles.container}>
07                 <Button title='调用原生组件' onPress={() => {
08                     if (Platform.OS === 'ios') {
09                         NativeModules.Communication.presentView
                            ControllerFromReactNative('12345');
10                     } else if (Platform.OS === 'android') {
11                         NativeModules.Communication.startActivityFrom
                            ReactNative('CommunicationActivity1', '12345');
12                     }
13                 }}/>
14             </View>
15         );
16     }
17 }
18
19 // 这里省略了没有修改的代码

```

(5) 这里 React Native 需要打开名为 CommunicationActivity1 的页面, 因此还需要在 Android 工程中添加该页面, 新建 CommunicationActivity1 类, 该类继承自 Activity, 添加代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class CommunicationActivity1 extends Activity {
04
05     @Override
06     protected void onCreate(Bundle savedInstanceState) {
07         super.onCreate(savedInstanceState);

```

```

08     setContentView(R.layout.communication_activity1);
09
10     Intent intent = getIntent();
11     if (intent != null) {
12         String params = intent.getStringExtra("params");
13         if (params != null) {
14             Toast.makeText(this, "从 React Native 传来的数据是: " +
15                 params, Toast.LENGTH_SHORT).show();
16         }
17     }
18
19 }

```

⚠注意: Android 开发中添加的 Activity 必须要在 AndroidManifest.xml 文件中注册 (<activity android:name=".CommunicationActivity1"/>), 否则将找不到该 Activity。

上述 CommunicationActivity1 中使用的布局文件 communication_activity1.xml 代码如下:

```

01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03     android:orientation="vertical" android:layout_width="match_parent"
04     android:layout_height="match_parent">
05
06     <TextView
07         android:id="@+id/textView"
08         android:layout_width="match_parent"
09         android:layout_height="wrap_content"
10         android:text="原生界面" />
11
12 </LinearLayout>

```

(6) 重新编译和运行 Android 应用, 打开“更多”页面, 单击“调用原生组件”按钮, 此时就打开了一个原生界面并显示 React Native 传递的参数, 效果如图 8.20 所示。

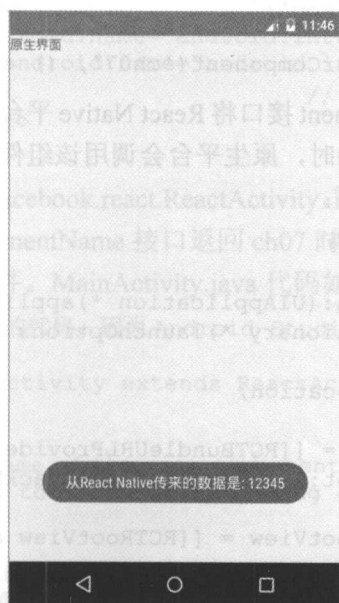



图 8.20 React Native 平台调用 Android 原生页面

 提示：上述例子涉及了较多的原生知识和代码，限于篇幅，这里没有一一详细介绍。感兴趣的读者，可以参考 iOS/Android 开发的相关书籍和教程。

8.6 原生平台调用 React Native 组件

本节接着介绍原生平台如何调用 React Native 组件。其实，我们很早就接触并了解了原生平台调用 React Native 组件的方式，只是读者可能没有留意到。

例如，React Native 应用根组件的注册和调用，原理如图 8.21 所示。

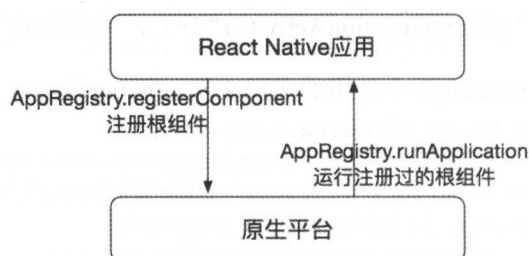


图 8.21 注册 React Native 应用的根组件

8.6.1 iOS 平台调用 React Native 组件

对于 iOS 平台，在 React Native 应用中首先注册根组件 ch07，index.ios.js 代码如下：

```

01 import React, {Component} from 'react';
02 import {AppRegistry} from 'react-native';
03 import app from './app';
04
05 AppRegistry.registerComponent('ch07', () => app);
  
```

AppRegistry.registerComponent 接口将 React Native 平台的根组件 ch07 注册到了原生平台上。当 React Native 应用启动时，原生平台会调用该组件，iOS 工程中调用 React Native 组件的 AppDelegate.m 代码如下：

```

01 // 这里省略了无关的代码
02
03 - (BOOL)application:(UIApplication *)application didFinishLaunching
    WithOptions:(NSDictionary *)launchOptions
    {
04     NSURL *jsCodeLocation;
05
06     jsCodeLocation = [[RCTBundleURLProvider sharedSettings] jsBundle
        URLForBundleRoot:@"index.ios" fallbackResource:nil];
07
08     RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:
        jsCodeLocation
                                moduleName:@"ch07"
                                initialProperties:nil
                                launchOptions:launchOptions];
  
```

```

09     rootView.backgroundColor = [[UIColor alloc] initWithRed:1.0f green:
10     1.0f blue:1.0f alpha:1];
11     self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].
12     bounds];
13     UIViewController *rootViewController = [UIViewController new];
14     rootViewController.view = rootView;
15     self.window.rootViewController = rootViewController;
16     [self.window makeKeyAndVisible];
17     return YES;
18 }
19 // 这里省略了无关的代码

```

8.6.2 Android 平台调用 React Native 组件

对于 Android 平台, 在 React Native 应用中首先注册根组件 ch07, index.android.js 代码如下:

```

01 import React, {Component} from 'react';
02 import {AppRegistry} from 'react-native';
03 import app from './app';
04
05 AppRegistry.registerComponent('ch07', () => app);

```

AppRegistry.registerComponent 接口将 React Native 平台的根组件 ch07 注册到了原生平台上。当 React Native 应用启动时, 原生平台首先会打开 MainActivity, AndroidManifest.xml 文件中的配置方法如下:

```

01 <activity
02     android:name=".MainActivity"
03     android:label="@string/app_name"
04     android:configChanges="keyboard|keyboardHidden|orientation|
05     screenSize">
06     <intent-filter>
07         <action android:name="android.intent.action.MAIN" />
08         <category android:name="android.intent.category.LAUNCHER" />
09         // 应用启动打开 MainActivity
10     </intent-filter>
11 </activity>

```

MainActivity 继承自 com.facebook.react.ReactActivity, ReactActivity 封装了 React Native 组件, 因此, 当 getMainComponentName 接口返回 ch07 时, 此时 MainActivity 打开的就是 React Native 中注册的 ch07 组件。MainActivity.java 代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class MainActivity extends ReactActivity {
04
05     /**
06     * Returns the name of the main component registered from JavaScript.
07     * This is used to schedule rendering of the component.
08     */
09     @Override {
10     protected String getMainComponentName()
11     {
12         return "ch07";
13     }
14 }

```

通过上述原生平台打开 React Native 根组件的过程可知, React Native 应用的启动流程如下:

- 首先还是启动原生平台的应用。
- 然后原生平台再调用注册的根组件。

8.7 小 结

目前,很多 App 都采用了 React Native 与原生代码混编的形式,虽然让读者学起来比较头疼,比如是否只学习 React Native 就够了,还是要多学习另外两种平台的开发,这样岂不是增加了很多负担?读者也可以只学习本章介绍的简单的原生功能,不过案例实现的手法就没有那么多了。接下来的第9章中还会学习更多的原生功能,希望读者能够继续保持学习的动力。

第9章 React Native 与原生平台混合编程（2）

原生功能除了访问设备、访问手机相册，其实还可以照相。同时，React Native 除了能与原生页面交互，还可以对整个项目进行支持，这些内容本章都会涉及。

本章主要内容有：

- 使用相机拍摄照片。
- 添加图片选择提示框。
- 重构图片选择库。
- 向 iOS 项目中添加 React Native 支持。
- 向 Android 项目中添加 React Native 支持。

9.1 使用相机拍摄图片

想要使用设备的图片资源，除了从相册读取外，还可以使用设备的相机拍摄照片。React Native 代码仍然基于 ImagePicker 组件，因此这里我们直接编写原生平台的实现。

9.1.1 使用 iOS 相机拍摄

首先添加并实现 launchCamera 接口，修改 ImagePicker.m 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 RCT_EXPORT_METHOD(launchCamera:(NSDictionary *)options callback:(RCT
  ResponseSenderBlock)callback) {
04     self.options = [NSMutableDictionary dictionaryWithDictionary: options];
05     self.callback = callback;
06
07     // 创建 UIImagePickerController
08     self.picker = [[UIImagePickerController alloc] init];
09     #if TARGET_IPHONE_SIMULATOR
10     self.callback(@[@"error": @"Camera not available on simulator"]);
11     return;
12     #else
13     self.picker.sourceType = UIImagePickerControllerSourceTypeCamera;
14     #endif
15     self.picker.mediaTypes = @[(NSString *)kUTTypeImage];
16     self.picker.modalPresentationStyle = UIModalPresentationCurrent
      Context;
17     self.picker.delegate = self;
18
19     // 获取相机权限
20     [self checkCameraPermissions:^(BOOL granted) {
```

```

21         if (!granted) {
22             self.callback(@[[@{"error": @"Camera permissions not
23                 granted"}]]);
24             return;
25         }
26         // 打开 UIImagePickerController
27         dispatch_async(dispatch_get_main_queue(), ^{
28             UIViewController *root = [[[UIApplication sharedApplication]
29                 delegate] window] rootViewController];
30             while (root.presentedViewController != nil) {
31                 root = root.presentedViewController;
32             }
33             [root presentViewController:self.picker animated:YES completion:
34                 nil];
35         });
36     }
37     // 这里省略了没有修改的代码
38
39     - (void)checkCameraPermissions:(void(^)(BOOL granted))callback {
40         AVAuthorizationStatus status = [AVCaptureDevice authorization
41             StatusForMediaType:AVMediaTypeVideo];
42         if (status == AVAuthorizationStatusAuthorized) {
43             callback(YES);
44             return;
45         } else if (status == AVAuthorizationStatusNotDetermined){
46             [AVCaptureDevice requestAccessForMediaType:AVMediaTypeVideo
47                 completionHandler:^(BOOL granted) {
48                     callback(granted);
49                     return;
50                 }];
51         } else {
52             callback(NO);
53         }
54     }
55     // 这里省略了没有修改的代码

```

接着再调整 `ImagePicker` 组件的相应逻辑：将调用的 `launchImagePicker` 接口改为 `launchCamera`，修改 `ImagePicker.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class ImagePicker extends Component {
04     // 这里省略了没有修改的代码
05
06     _selectPhotoTapped = () => {
07         const options = {
08             quality: 1.0,
09             maxWidth: 500,
10             maxHeight: 500
11         };
12
13         NativeModules.ImagePicker.launchCamera(options, (response) => {
14             // 启动相机
15
16             if (response.didCancel) {
17                 console.log('取消选择图片');
18             } else if (response.error) {

```

```

17         console.log('选择图片错误: ', response.error);
18     } else {
19         let source;
20         if (Platform.OS === 'ios') {
21             source = {
22                 uri: response.uri.replace('file://', '')
23             };
24         } else if (Platform.OS === 'android') {
25             source = {
26                 uri: response.uri
27             };
28         }
29         this.setState({avatarSource: source});
30     }
31     });
32 }
33
34 // 这里省略了没有修改的代码
35 }
36
37
38 // 这里省略了没有修改的代码

```

在编译和运行应用之前需要注意的是, 由于 iOS 模拟器不支持摄像头, 所以这里需要使用 iPhone 真机来验证上述例子, 首先连接 iPhone 真机, 然后在 Xcode 中选择已连接的 iPhone 真机, 效果如图 9.1 所示。

重新编译和运行应用, 打开“更多”页面, 单击“选择图片”按钮, 然后使用相机拍摄一张图片, 选择右侧的 Use Photo 按钮, 效果如图 9.2 所示。

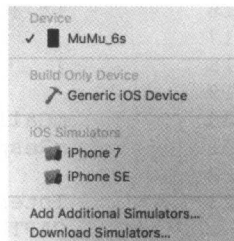


图 9.1 Xcode 使用 iPhone 真机



图 9.2 iOS 平台使用相机拍摄图片

9.1.2 使用 Android 相机拍摄

首先还是添加并实现 `launchCamera` 接口，修改 `ImagePickerModule.java` 代码如下：

```

01 // 这里省略了导入文件的代码，因为 Android Studio 会自动导入所依赖的模块
02
03 public class ImagePickerModule extends ReactContextBaseJavaModule
    implements ActivityEventListener {
04     static final int REQUEST_LAUNCH_IMAGE_CAPTURE = 13001;
05     static final int REQUEST_LAUNCH_IMAGE_LIBRARY = 13002;
06     private final ReactApplicationContext mReactContext;
07     private Callback mCallback;
08     private WritableMap mResponse;
09     private Uri mCameraCaptureURI;
10
11     // 这里省略了没有修改的代码
12
13     @ReactMethod
14     public void launchCamera(final ReadableMap options, final Callback
        callback) {
15         // 判断设备是否支持相机
16         if (!isCameraAvailable()) {
17             return;
18         }
19
20         mCallback = callback;
21         mResponse = Arguments.createMap();
22
23         Intent cameraIntent = new Intent(MediaStore.ACTION_IMAGE_
            CAPTURE);
24         mCameraCaptureURI = Uri.fromFile(createNewFile());
25         cameraIntent.putExtra(MediaStore.EXTRA_OUTPUT, mCameraCaptureURI);
26
27         try {
28             // 打开相机，并存储拍摄的图片
29             int requestCode = REQUEST_LAUNCH_IMAGE_CAPTURE;
30             Activity currentActivity = getCurrentActivity();
31             currentActivity.startActivityForResult(cameraIntent,
                requestCode);
32         } catch (ActivityNotFoundException e) {
33             e.printStackTrace();
34             if (mCallback != null) {
35                 mResponse.putString("error", "Cannot launch camera");
36                 mCallback.invoke(mResponse);
37                 mCallback = null;
38             }
39         }
40     }
41
42     // 这里省略了具体的辅助接口实现
43 }

```

同时还需要更新对用户选择图片操作的处理，修改 `ImagePickerModule.java` 文件中的 `onActivityResult` 代码如下：

```

01 // 这里省略了导入文件的代码，因为 Android Studio 会自动导入所依赖的模块
02
03 public class ImagePickerModule extends ReactContextBaseJavaModule

```

```

implements ActivityEventListener {
04    // 这里省略了上述已经描述的代码
05
06    public void onActivityResult(Activity activity, int requestCode,
        int resultCode, Intent data) {
07        mResponse = Arguments.createMap();
08
09        // 用户取消
10        if (resultCode != Activity.RESULT_OK) {
11            mResponse.putBoolean("didCancel", true);
12            mCallback.invoke(mResponse);
13            mCallback = null;
14            return;
15        }
16
17        // 获取图片 uri
18        Uri uri;
19        switch (requestCode) {
20            case REQUEST_LAUNCH_IMAGE_CAPTURE:
21                uri = mCameraCaptureURI;
22                break;
23            case REQUEST_LAUNCH_IMAGE_LIBRARY:
24                uri = data.getData();
25                break;
26            default:
27                uri = null;
28        }
29        String realPath = getRealPathFromURI(uri);
30
31        if (!TextUtils.isEmpty(realPath)) {
32            // 解码图片
33            Options options = new Options();
34            options.inJustDecodeBounds = true;
35            BitmapFactory.decodeFile(realPath, options);
36
37            // 回调函数
38            mResponse.putString("uri", uri.toString());
39            mResponse.putString("path", realPath);
40            mCallback.invoke(mResponse);
41            mCallback = null;
42        } else {
43            if (mCallback != null) {
44                mResponse.putString("error", "Cannot launch photo library");
45                mCallback.invoke(mResponse);
46                mCallback = null;
47            }
48        }
49    }
50
51    // 这里省略了具体的辅助接口实现
52 }

```

上述代码中涉及的辅助接口 `isCameraAvailable` 和 `createNewFile` 代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class ImagePickerModule extends ReactContextBaseJavaModule
    implements ActivityEventListener {
04     // 这里省略了上述已经描述的代码
05
06     private boolean isCameraAvailable() {

```

```

07         return mReactContext.getPackageManager().hasSystemFeature(Package
Manager.FEATURE_CAMERA)
08         || mReactContext.getPackageManager().hasSystemFeature(Package
Manager.FEATURE_CAMERA_ANY);
09     }
10
11     private File createNewFile() {
12         String filename = "image-" + UUID.randomUUID().toString() +
".jpg";
13         File path = Environment.getExternalStoragePublicDirectory
(Environment.DIRECTORY_PICTURES);
14         File f = new File(path, filename);
15         try {
16             path.mkdirs();
17             f.createNewFile();
18         } catch (IOException e) {
19             e.printStackTrace();
20         }
21         return f;
22     }
23
24 }

```

最后读者容易忘记的是，由于使用到了 Android 设备的相机和存储功能，所以还需要请求权限，在 Android 工程的 `AndroidManifest.xml` 文件中添加如下配置：

```

<uses-permission android:name="android.permission.CAMERA" />
<uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

```

重新编译和运行应用，打开“更多”页面，单击“选择图片”按钮，然后使用相机拍摄一张图片，效果如图 9.3 所示。

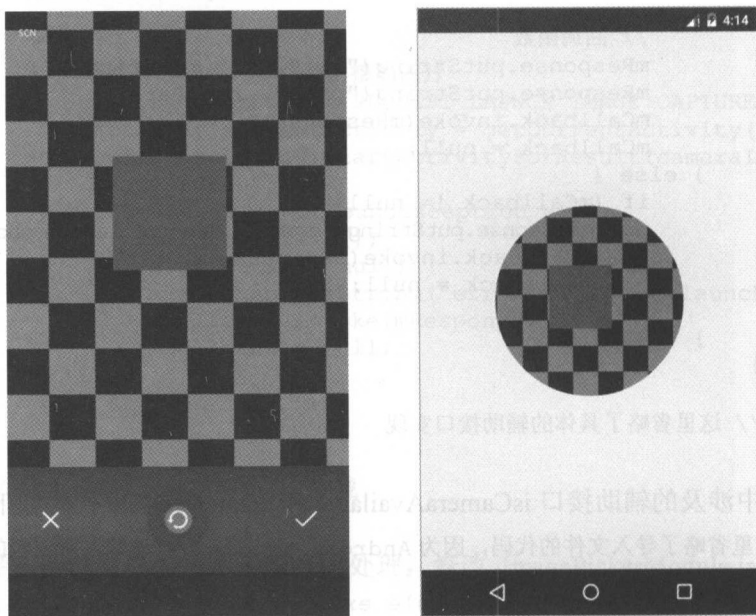


图 9.3 Android 平台使用相机拍摄图片

9.2 添加图片选择提示框

第8章的读取相册和9.1节的使用相机拍摄图片的内容组成了图片库的基本功能。要实现完整的图片库，只需要再添加一个图片选择提示框。

9.2.1 iOS 平台的提示

对于iOS平台，首先添加并实现 `launchImageLibrary` 接口，修改 `ImagePicker.m` 代码如下：

```

01 // 这里省略了包含的头文件
02
03 @import MobileCoreServices;
04
05 @interface ImagePicker ()
06
07 @property (nonatomic, retain) NSMutableDictionary *options, *response;
08 @property (nonatomic, strong) RCTResponseSenderBlock callback;
09 @property (nonatomic, strong) UIImagePickerController *picker;
10 @property (nonatomic, strong) UIAlertController *alertController;
11                                     // 添加 UIAlertController
12 @end
13
14 @implementation ImagePicker
15
16 RCT_EXPORT_METHOD(launchImageLibrary:(NSDictionary *)options callback:
17 (RCTResponseSenderBlock)callback) {
18     self.callback = callback;
19
20     // 创建 UIAlertController
21     self.alertController = [UIAlertController alertControllerWithTitle:
22 @"照片库" message:nil preferredStyle:UIAlertControllerStyle
23 ActionSheet];
24
25     UIAlertAction *cancelAction = [UIAlertAction actionWithTitle:@"取消"
26 style:UIAlertActionStyleCancel handler:^(UIAlertAction * action) {
27     self.callback(@[@"didCancel": YES]);
28 }];
29     [self.alertController addAction:cancelAction];
30
31     UIAlertAction *takePhotoAction = [UIAlertAction actionWithTitle:@"相册"
32 style:UIAlertActionStyleDefault handler:^(UIAlertAction *
33 action) {
34     [self launchImagePicker:options callback:callback];
35 }];
36     [self.alertController addAction:takePhotoAction];
37
38     UIAlertAction *chooseFromLibraryAction = [UIAlertAction action
39 withTitle:@"相机" style:UIAlertActionStyleDefault handler:^(
40 UIAlertAction * action) {
41     [self launchCamera:options callback:callback];
42 }];
43     [self.alertController addAction:chooseFromLibraryAction];
44
45     // 打开 UIAlertController
46     dispatch_async(dispatch_get_main_queue(), ^{

```

```

36      UINavigationController *root = [[[[UIApplication sharedApplication]
37      delegate] window] rootViewController];
38      [root presentViewController:self.alertController animated:YES
39      completion:nil];
40  });
41  // 这里省略了没有修改的代码

```

接着将 React Native 组件 ImagePicker 调用的接口改为 launchImageLibrary，修改 ImagePicker.js 代码如下：

```

01  // 这里省略了没有修改的代码
02
03  export default class ImagePicker extends Component {
04      // 这里省略了没有修改的代码
05
06      _selectPhotoTapped = () => {
07          const options = {
08              quality: 1.0,
09              maxWidth: 500,
10              maxHeight: 500
11          };
12
13          NativeModules.ImagePicker.launchImageLibrary(options, (response)
14          => { // 启动图片库
15              if (response.didCancel) {
16                  console.log('取消选择图片');
17              } else if (response.error) {
18                  console.log('选择图片错误: ', response.error);
19              } else {
20                  let source;
21                  if (Platform.OS === 'ios') {
22                      source = {
23                          uri: response.uri.replace('file://', '')
24                      };
25                  } else if (Platform.OS === 'android') {
26                      source = {
27                          uri: response.uri
28                      };
29                  }
30                  this.setState({avatarSource: source});
31              }
32          });
33
34      // 这里省略了没有修改的代码
35  }
36
37
38  // 这里省略了没有修改的代码

```

重新编译和运行应用，打开“更多”页面，单击“选择图片”按钮，会打开图片选择提示框，效果如图 9.4 所示。

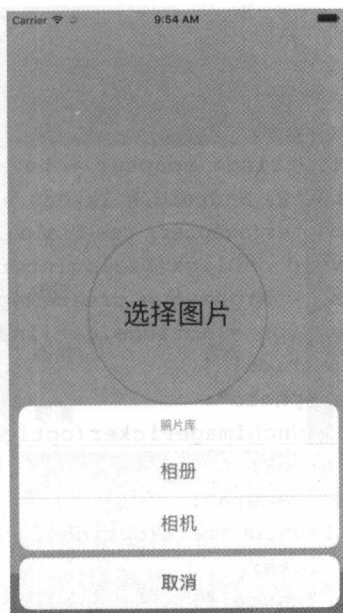


图 9.4 iOS 图片选择提示框

9.2.2 Android 平台的提示

首先还是添加并实现 `launchImageLibrary` 接口, 修改 `ImagePickerModule.java` 代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class ImagePickerModule extends ReactContextBaseJavaModule
    implements ActivityEventListener {
04     // 这里省略了没有修改的代码
05
06     @ReactMethod
07     public void launchImageLibrary(final ReadableMap options, final
        Callback callback) {
08         Activity currentActivity = getCurrentActivity();
09         if (currentActivity == null) {
10             return;
11         }
12
13         // 创建 AlertDialog.Builder
14         AlertDialog.Builder builder = new AlertDialog.Builder
            (currentActivity, android.R.style.Theme_Holo_Light_Dialog);
15         builder.setTitle("相片库");
16
17         // 设置 AlertDialog 的 Adapter
18         final List<String> titles = new ArrayList<String>();
19         final List<String> actions = new ArrayList<String>();
20         titles.add("相册");
21         actions.add("photo");

```

```
22         titles.add("相机");
23         actions.add("camera");
24         titles.add("取消");
25         actions.add("cancel");
26         ArrayAdapter<String> adapter = new ArrayAdapter<String>
27             (currentActivity, android.R.layout.select_dialog_item, titles);
28         builder.setAdapter(adapter, new DialogInterface.OnClickListener() {
29             public void onClick(DialogInterface dialog, int index) {
30                 mResponse = Arguments.createMap();
31                 String action = actions.get(index);
32                 switch (action) {
33                     case "photo":
34                         launchImagePicker(options, callback);
35                         break;
36                     case "camera":
37                         launchCamera(options, callback);
38                         break;
39                     case "cancel":
40                         mResponse.putBoolean("didCancel", true);
41                         callback.invoke(mResponse);
42                         break;
43                     default:
44                         break;
45                 }
46             });
47
48         // 创建 AlertDialog
49         final AlertDialog dialog = builder.create();
50         dialog.setOnCancelListener(new DialogInterface.OnCancelListener() {
51             @Override
52             public void onCancel(DialogInterface dialog) {
53                 mResponse = Arguments.createMap();
54                 dialog.dismiss();
55                 mResponse.putBoolean("didCancel", true);
56                 callback.invoke(mResponse);
57             }
58         });
59
60         // 显示 AlertDialog
61         dialog.show();
62     }
63
64     // 这里省略了没有修改的代码
65 }
```

重新编译和运行应用，打开“更多”页面，单击“选择图片”按钮，会打开图片选择提示框，效果如图 9.5 所示。

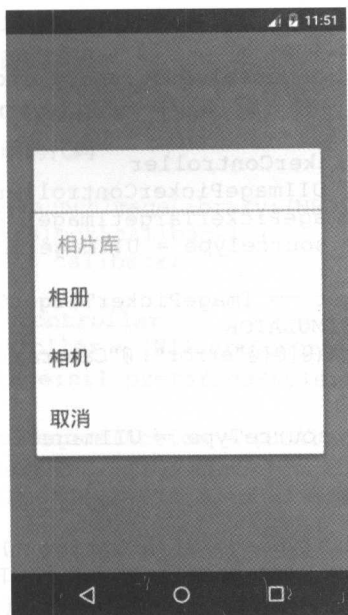


图 9.5 Android 图片选择提示框

9.3 重构图片选择库

至此已经开发了一个完整的图片选择库的功能，包括：

- 图片选择提示框。
- 从相册选择图片。
- 用相机拍摄图片。

细心的读者可能会发现代码中存在这样的问题：“从相册选择图片”和“用相机拍摄图片”的大部分实现是相同的，因此最后需要重构图片选择库的代码。

9.3.1 iOS 平台的重构

首先定义一个枚举类型 `ImagePickerTarget`，用于表示用户选择的是相册还是相机，添加 `ImagePicker.m` 代码如下：

```
01 // 这里省略了包含的头文件
02
03 typedef NSInteger ImagePickerTarget {
04     ImagePickerTargetImage = 0,
05     ImagePickerTargetCamera
06 };
07
08 // 这里省略了没有修改的代码
```

然后修改接口 `launchImagePicker` 的参数和实现，修改 `ImagePicker.m` 代码如下：

```
01 // 这里省略了没有修改的代码
02
```

```
03 RCT_EXPORT_METHOD(launchImagePicker:(ImagePickerTarget)target options:
   (NSDictionary *)options) {
04     self.options = [NSMutableDictionary dictionaryWithDictionary:
       options];
05
06     // 创建 UIImagePickerControllerController
07     self.picker = [[UIImagePickerController alloc] init];
08     if (target == ImagePickerTargetImage) {
09         self.picker.sourceType = UIImagePickerControllerSourceType
           PhotoLibrary;
10     } else if (target == ImagePickerTargetCamera) {
11 #if TARGET_IPHONE_SIMULATOR
12         self.callback(@[@"error": @"Camera not available on simulator"]);
13         return;
14 #else
15         self.picker.sourceType = UIImagePickerControllerSourceType
           Camera;
16 #endif
17     }
18     self.picker.mediaTypes = @[(NSString *)kUTTypeImage];
19     self.picker.modalPresentationStyle = UIModalPresentationCurrent
       Context;
20     self.picker.delegate = self;
21
22     void (^showPickerViewController)() = ^void() {
23         // 打开 UIImagePickerController
24         dispatch_async(dispatch_get_main_queue(), ^{
25             UIViewController *root = [[[UIApplication sharedApplication]
               delegate] window] rootViewController];
26             while (root.presentedViewController != nil) {
27                 root = root.presentedViewController;
28             }
29             [root presentViewController:self.picker animated:YES completion:
               nil];
30         });
31     };
32
33     if (target == ImagePickerTargetImage) {
34         // 获取相册权限
35         [self checkCameraPermissions:^(BOOL granted) {
36             if (!granted) {
37                 self.callback(@[@"error": @"Camera permissions not
                   granted"]);
38                 return;
39             }
40
41             showPickerViewController();
42         }];
43     } else if (target == ImagePickerTargetCamera) {
44         // 获取相机权限
45         [self checkPhotosPermissions:^(BOOL granted) {
46             if (!granted) {
47                 self.callback(@[@"error": @"Photo library permissions
                   not granted"]);
48                 return;
49             }
50
51             showPickerViewController();
52         }];
53     }
54 }
```

```

55
56 // 这里省略了没有修改的代码

同时, 修改接口 launchImageLibrary 中的实现, 修改 ImagePicker.m 代码如下:

01 // 这里省略了没有修改的代码
02
03 RCT_EXPORT_METHOD(launchImageLibrary:(NSDictionary *)options callback:
(RCTResponseSenderBlock)callback) {
04     self.callback = callback;
05
06     // 创建 UIAlertController
07     self.alertController = [UIAlertController alertControllerWithTitle:
@"照片库" message:nil preferredStyle:UIAlertControllerStyleAction
Sheet];
08     UIAlertAction *cancelAction = [UIAlertAction actionWithTitle:@"取
消" style:UIAlertActionStyleCancel handler:^(UIAlertAction * action) {
09         self.callback(@[@"didCancel": @YES]);
10     }];
11     [self.alertController addAction:cancelAction];
12     UIAlertAction *takePhotoAction = [UIAlertAction actionWithTitle:@
"相册" style:UIAlertActionStyleDefault handler:^(UIAlertAction * action) {
13         [self launchImagePicker:ImagePickerTargetImage options:options];
14     }];
15     [self.alertController addAction:takePhotoAction];
16     UIAlertAction *chooseFromLibraryAction = [UIAlertAction action
WithTitle:@"相机" style:UIAlertActionStyleDefault handler:^(UIAlert
Action * action) {
17         [self launchImagePicker:ImagePickerTargetCamera options:options];
18     }];
19     [self.alertController addAction:chooseFromLibraryAction];
20
21     // 打开 UIAlertController
22     dispatch_async(dispatch_get_main_queue(), ^{
23         UIViewController *root = [[[UIApplication sharedApplication]
delegate] window] rootViewController];
24         [root presentViewController:self.alertController animated:YES
completion:nil];
25     });
26 }
27
28 // 这里省略了没有修改的代码

```

最后删除废弃的接口 `launchCamera` 以及相关代码。

此时重新编译和运行应用, 打开“更多”页面, 单击“选择图片”按钮, 验证图片选择库的功能仍然是完整的。

9.3.2 Android 平台的重构

首先在 `ImagePickerModule` 类中添加一个属性 `mImagePickerTargetType`, 用于表示当前选择的是相册还是相机, 添加 `ImagePickerModule.java` 代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class ImagePickerModule extends ReactContextBaseJavaModule
implements ActivityEventListener {
04     static final int REQUEST_LAUNCH_IMAGE_CAPTURE = 13001;
05     static final int REQUEST_LAUNCH_IMAGE_LIBRARY = 13002;

```

```

06     static final int IMAGE_PICKER_TARGET_IMAGE = 0;
07     static final int IMAGE_PICKER_TARGET_CAMERA = 1;
08     private final ReactApplicationContext mReactContext;
09     private Callback mCallback;
10     private WritableMap mResponse;
11     private Uri mCameraCaptureURI;
12     private int mImagePickerTargetType;
13
14     // 这里省略了没有修改的代码
15 }

```

然后修改接口 `launchImagePicker` 的参数和实现，修改 `ImagePickerModule.java` 代码：

```

01 // 这里省略了导入文件的代码，因为 Android Studio 会自动导入所依赖的模块
02
03 public class ImagePickerModule extends ReactContextBaseJavaModule
04     implements ActivityEventListener {
05
06     // 这里省略了没有修改的代码
07
08     @ReactMethod
09     public void launchImagePicker(final ReadableMap options, final
10         Callback callback) {
11
12         // 判断设备是否支持相机
13         if (mImagePickerTargetType == IMAGE_PICKER_TARGET_CAMERA
14             && !isCameraAvailable()) {
15             return;
16         }
17
18         mCallback = callback;
19         mResponse = Arguments.createMap();
20
21         Intent libraryIntent = null;
22         if (mImagePickerTargetType == IMAGE_PICKER_TARGET_IMAGE) {
23             libraryIntent = new Intent(Intent.ACTION_PICK, MediaStore.
24                 Images.Media.EXTERNAL_CONTENT_URI);
25         } else if (mImagePickerTargetType == IMAGE_PICKER_TARGET_CAMERA) {
26             libraryIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
27             mCameraCaptureURI = Uri.fromFile(createNewFile());
28             libraryIntent.putExtra(MediaStore.EXTRA_OUTPUT,
29                 mCameraCaptureURI);
30         }
31
32         try {
33             // 打开相册或相机
34             int requestCode = mImagePickerTargetType == IMAGE_PICKER_
35                 TARGET_IMAGE ?
36                 REQUEST_LAUNCH_IMAGE_LIBRARY : REQUEST_LAUNCH_IMAGE_
37                 CAPTURE;
38             Activity currentActivity = getCurrentActivity();
39             currentActivity.startActivityForResult(libraryIntent,
40                 requestCode);
41         } catch (ActivityNotFoundException e) {
42             e.printStackTrace();
43             if (mCallback != null) {
44                 mResponse.putString("error", "Cannot launch photo
45                     library");
46                 mCallback.invoke(mResponse);
47                 mCallback = null;
48             }
49         }
50     }
51 }

```

```

40
41     // 这里省略了没有修改的代码
42 }

```

同时, 修改接口 `launchImageLibrary` 中的实现, 修改 `ImagePickerModule.java` 代码如下:


```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class ImagePickerModule extends ReactContextBaseJavaModule
    implements ActivityEventListener {
04     // 这里省略了没有修改的代码
05
06     @ReactMethod
07     public void launchImageLibrary(final ReadableMap options, final
        Callback callback) {
08         // 这里省略了没有修改的代码
09         ArrayAdapter<String> adapter = new ArrayAdapter<String>
            (currentActivity, android.R.layout.select_dialog_item, titles);
10         builder.setAdapter(adapter, new DialogInterface.OnClickListener() {
11             public void onClick(DialogInterface dialog, int index) {
12                 mResponse = Arguments.createMap();
13                 String action = actions.get(index);
14                 switch (action) {
15                     case "photo":
16                         mImagePickerTargetType = IMAGE_PICKER_TARGET_IMAGE;
17                         launchImagePicker(options, callback);
18                         break;
19                     case "camera":
20                         mImagePickerTargetType = IMAGE_PICKER_TARGET_CAMERA;
21                         launchImagePicker(options, callback);
22                         break;
23                     case "cancel":
24                         mResponse.putBoolean("didCancel", true);
25                         callback.invoke(mResponse);
26                         break;
27                     default:
28                         break;
29                 }
30             }
31         // 这里省略了没有修改的代码
32     }
33
34     // 这里省略了没有修改的代码
35 }

```

最后删除废弃的接口 `launchCamera` 以及相关代码。

此时重新编译和运行应用, 打开“更多”页面, 单击“选择图片”按钮, 验证图片选择库的功能仍然是完整的。

 **提示:** 本书为了讲解和讨论的需要, 自己动手编写了图片选择库的基本功能。在 React Native 实际开发中, 除了自己编写原生代码扩展应用的功能之外, 尽量还是使用成熟的第三方库, 例如, 图片选择库可以使用 `react-native-image-picker` (<https://github.com/marcshilling/react-native-image-picker>)。

9.4 向 iOS 项目中添加 React Native 支持

之前为 React Native 应用扩展原生功能，还是以 React Native 开发为主导，然而很多项目已经使用了原生开发，现在考虑逐步引入 React Native，因此就会遇到如何向已有的原生项目添加 React Native 支持的问题。本节先介绍如何向 iOS 项目中添加 React Native 支持。

9.4.1 新建 iOS 项目

首先新建一个原生项目。打开 Xcode，选择 Create a new Xcode Project，然后在项目模板中选择 Single View Application 项，效果如图 9.6 所示。

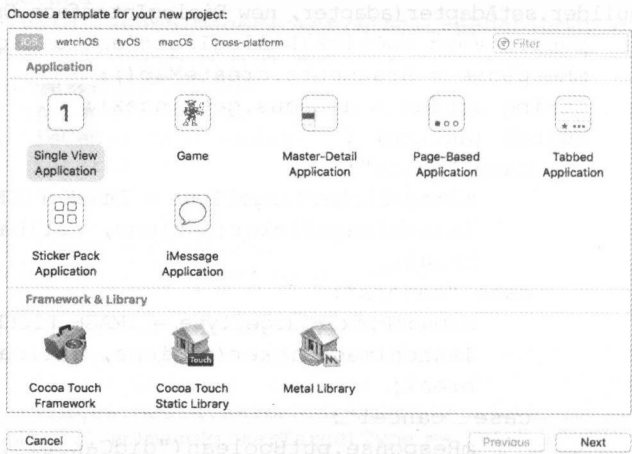


图 9.6 选择 iOS 原生项目模板

接着配置项目的项目名称、组织名称、编程语言以及设备等信息，效果如图 9.7 所示。



图 9.7 配置 iOS 原生项目信息

9.4.2 新建 React Native 项目

为了向上述 iOS 原生项目中添加 React Native 支持，通常先新建一个同名的 React Native 项目。

(1) 新建 React Native 项目命令如下：

```
react-native init AddReactNativeToIOS // 新建 React Native 项目 AddReactNativeToIOS
cd AddReactNativeToIOS
npm install
```

(2) 删除 React Native 项目的 ios 目录，同时将 iOS 原生项目的目录名改为 ios 并复制至 React Native 项目下。

(3) 使用 Xcode 打开 ios/AddReactNativeToIOS.xcodeproj 文件。单击 Xcode 左侧项目一栏中底部的加号，效果如图 9.8 所示。

(4) 选择 Add Files to “AddReactNativeToIOS” 选项，然后找到该文件 node_modules/react-native/React/React.xcodeproj，单击“添加”按钮，效果如图 9.9 所示。

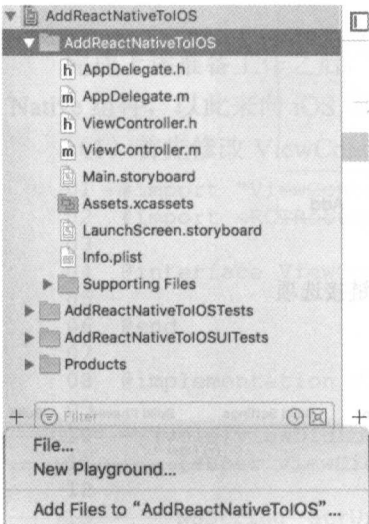


图 9.8 添加项目依赖

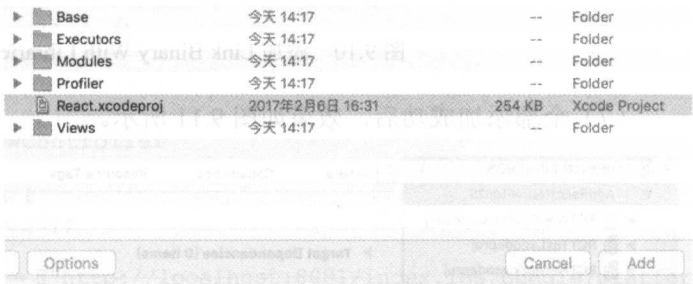


图 9.9 选择依赖的项目

(5) 把依赖的静态库加到链接选项中，打开 Xcode 工程 Build Phases 下的 Link Binary With Libraries 选项，单击加号，选择 libReact.a from ‘React’ target in ‘React’ project 项，效果如图 9.10 所示。

(6) 按照同样的方法，再添加其他的依赖如下：

```
node_modules/react-native/Libraries/Network/RCTNetwork.xcodeproj
node_modules/react-native/Libraries/Text/RCTText.xcodeproj
node_modules/react-native/Libraries/WebSocket/RCTWebSocket.xcodeproj
```

提示：这里添加的依赖只是 React Native 开发中用到的基本功能。如果想要在 React Native 开发中使用 Image 组件，就需要添加 RCTImage.xcodeproj 依赖。更多的依赖项目还有：

```
node_modules/react-native/Libraries/NativeAnimation/RCTAnimation.xcodeproj
node_modules/react-native/Libraries/ActionSheetIOS/RCTActionSheet.xcodeproj
node_modules/react-native/Libraries/Geolocation/RCTGeolocation.xcodeproj
node_modules/react-native/Libraries/Image/RCTImage.xcodeproj
node_modules/react-native/Libraries/LinkingIOS/RCTLinking.xcodeproj
node_modules/react-native/Libraries/Settings/RCTSettings.xcodeproj
node_modules/react-native/Libraries/Vibration/RCTVibration.xcodeproj
node_modules/react-native-vector-icons/RNVectorIcons.xcodeproj
```

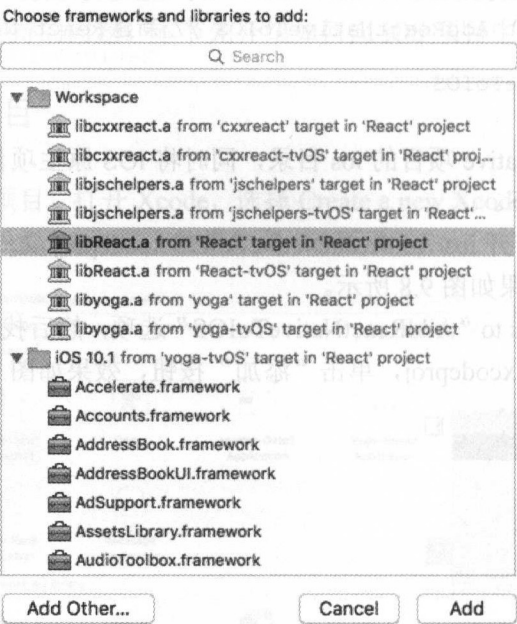


图 9.10 添加 Link Binary With Libraries 链接选项

(7) 全部添加成功后，效果如图 9.11 所示。

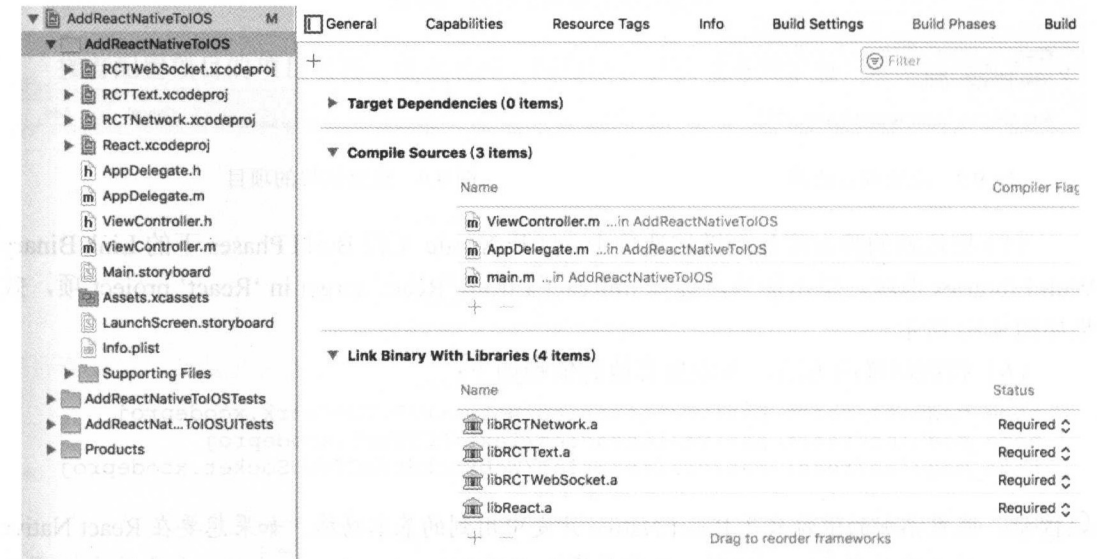


图 9.11 添加所有依赖后的 iOS 项目

(8) 最后还需要配置编译选项, 打开 Xcode 工程 Build Settings 下的 Other Link Flags 选项, 单击加号, 添加 `-ObjC` 和 `-l"stdc++"` 两项, 效果如图 9.12 所示。



图 9.12 配置 Other Link Flags 编译选项

9.4.3 在 iOS 页面打开 React Native 组件

完成上述准备工作之后, 就可以编写 iOS 原生项目代码, 让 iOS 页面能够打开 React Native 组件, 以此来向 iOS 项目中添加 React Native 支持。

(1) 首先修改 `ViewController.m` 代码如下:

```
01 #import "ViewController.h"
02 #import <RCTRootView.h>
03
04 @interface ViewController ()
05
06 @end
07
08 @implementation ViewController
09
10 - (void)viewDidLoad {
11     [super viewDidLoad];
12
13     NSString *strUrl = @"http://localhost:8081/index.ios.bundle?platform=
ios&dev=true";
14     NSURL *jsCodeLocation = [NSURL URLWithString:strUrl];
15 #ifndef DEBUG
16     jsCodeLocation = [[NSBundle mainBundle] URLForResource:@"main"
withExtension:@"jsbundle"];
17 #endif
18     RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:
jsCodeLocation moduleName:@"AddReactNativeToIOS" initialProperties:
nil launchOptions:nil];
19     rootView.frame = self.view.bounds;
20     [self.view addSubview:rootView];
21 }
22
23 @end
```

上述代码中的 `RCTRootView` 会将 React Native 中的组件封装成原生平台的视图。

(2) 此时, 编译 iOS 工程会出现 `'RCTRootView.h' file not found` 错误, 如图 9.13 所示。

为了解决找不到头文件的错误,继续配置相关编译选项,打开 Xcode 工程 Build Settings 下的 Header Search Paths 选项,单击加号,添加\$(SRCROOT)/../node_modules/react-native 项,并选择 recursive,效果如图 9.14 所示。



图 9.13 'RCTRootView.h' file not found 错误

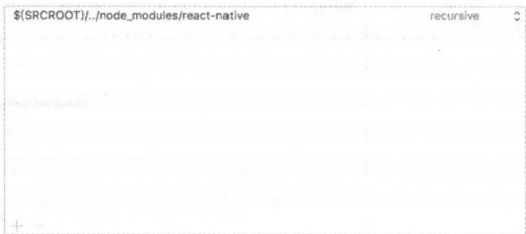


图 9.14 配置 Header Search Paths 编译选项

(3) 最后在编译和运行应用之前,还需要允许应用使用 HTTP 请求。打开 Xcode 工程的 Info 选项,添加 App Transport Security Settings 并设置类型为 Dictionary,然后在该描述下添加 Allow Arbitrary Loads 并设置类型为 Boolean 值为 YES,如图 9.15 所示。

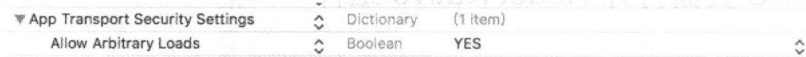



图 9.15 配置 App Transport Security Settings 项目描述

 **小知识:** 关于 iOS 开发 App Transport Security 的更多介绍,可以参考 Apple 官方文档 <https://developer.apple.com/library/content/releasenotes/General/WhatsNewIniOS/Articles/iOS9.html>。

终于完成了所有配置!此时,编译和运行 iOS 原生应用,就可以看到 React Native 组件显示在原生平台的页面中了,效果如图 9.16 所示。



图 9.16 React Native 组件显示在 iOS 原生页面中

9.5 向 Android 项目中添加 React Native 支持

前面讲解了 iOS 项目中添加 React Native，本节再来看看 Android 项目中的添加。

9.5.1 新建 Android 项目

首先新建一个原生项目。打开 Android Studio，选择 Start a new Android Studio project，配置项目的应用名称、组织名称、以及项目路径等信息，效果如图 9.17 所示。接着在之后所有的弹出框中都单击 Next 按钮，直到完成 Android 原生项目的创建。

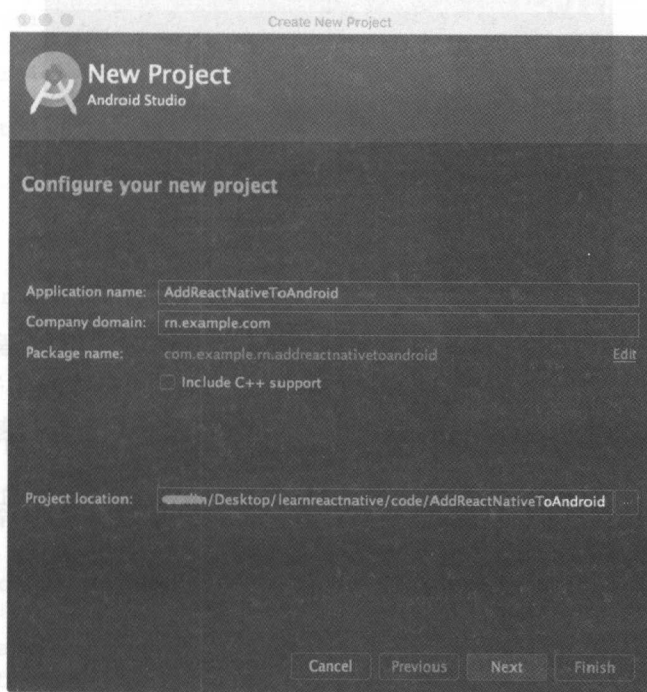


图 9.17 配置 Android 原生项目信息

9.5.2 新建 React Native 项目

为了向上述 Android 原生项目中添加 React Native 支持，通常先新建一个同名的 React Native 项目。新建 React Native 项目命令如下：

```
react-native init AddReactNativeToAndroid // 新建 React Native 项目 AddReact NativeToAndroid
cd AddReactNativeToAndroid
npm install
```

然后删除 React Native 项目的 android/app/src 目录，同时将 Android 原生项目的 src 目录复制至刚才删除的相应目录。

9.5.3 在 Android 页面打开 React Native 组件

完成上述准备工作之后，就可以编写 Android 原生项目代码，让 Android 页面能够打开 React Native 组件，以此来向 Android 项目中添加 React Native 支持。

(1) 新建 MainApplication 类，该类继承自 `android.app.Application` 并且实现 `com.facebook.react.ReactApplication` 接口，效果如图 9.18 所示。

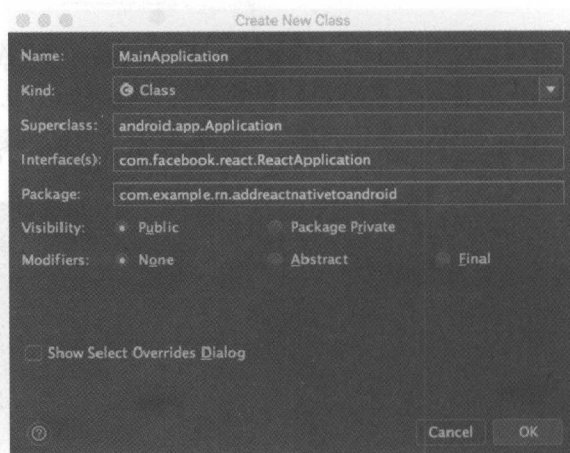


图 9.18 在 Android 工程中新建 MainApplication 类

(2) 添加 MainApplication.java 代码如下：

```

01 // 这里省略了导入文件的代码，因为 Android Studio 会自动导入所依赖的模块
02
03 public class MainApplication extends Application implements React
    Application {
04
05     private final ReactNativeHost mReactNativeHost = new React
        NativeHost(this) {
06         @Override
07         public boolean getUseDeveloperSupport() {
08             return BuildConfig.DEBUG;
09         }
10
11         @Override
12         protected List<ReactPackage> getPackages() {
13             return Arrays.<ReactPackage>asList(
14                 new MainReactPackage()
15             );
16         }
17     };
18
19     @Override
20     public ReactNativeHost getReactNativeHost() {
21         return mReactNativeHost;
22     }
23
24     @Override
25     public void onCreate() {
26         super.onCreate();
    
```

```

27     SoLoader.init(this, /* native exopackage */ false);
28 }
29
30 }

```

(3) 在 Android 工程的 AndroidManifest.xml 文件中注册 MainApplication:

```

react-native init AddReactNativeToAndroid // 新建 React Native 项目 AddReact
NativeToAndroid
<application
    android:name=".MainApplication"
    // 这里省略了没有修改的配置
</application>

```

(4) 在 MainActivity 中添加一个跳转到使用 React Native 组件页面的按钮, 修改 activity_main.xml 代码如下:

```

01 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
02     android:layout_width="match_parent"
03     android:layout_height="match_parent"
04     android:orientation="vertical">
05
06     <Button
07         android:id="@+id/button"
08         android:layout_width="match_parent"
09         android:layout_height="wrap_content"
10         android:text="打开使用 React Native 组件的页面"/>
11
12 </LinearLayout>

```

此时, 重新编译和运行应用, 页面的效果如图 9.19 所示。

(5) 准备好上述页面和功能后, 就可以添加一个新的使用 React Native 组件的 SecondActivity, 该类继承自 com.facebook.react.ReactActivity, 添加 SecondActivity.java 代码如下:

```

01 // 这里省略了导入文件的代码, 因为 Android Studio 会自动导入所依赖的模块
02
03 public class SecondActivity extends ReactActivity {
04
05     @Override
06     protected String getMainComponentName() {
07         return "AddReactNativeToAndroid";
08     }
09
10 }

```

(6) 不要忘记在 Android 工程的 AndroidManifest.xml 文件中注册 SecondActivity:

```
<activity android:name=".SecondActivity"></activity>
```

(7) 在编译和运行应用之前, 还需要请求网络权限, 在 Android 工程的 AndroidManifest.xml 文件中添加如下配置:

```
<uses-permission android:name="android.permission.INTERNET" />
```

终于完成了所有配置! 此时编译和运行 Android 原生应用, 单击“打开使用 REACT NATIVE 组件的页面”按钮, 就可以跳转到使用 React Native 组件的页面中了, 效果如图 9.20 所示。

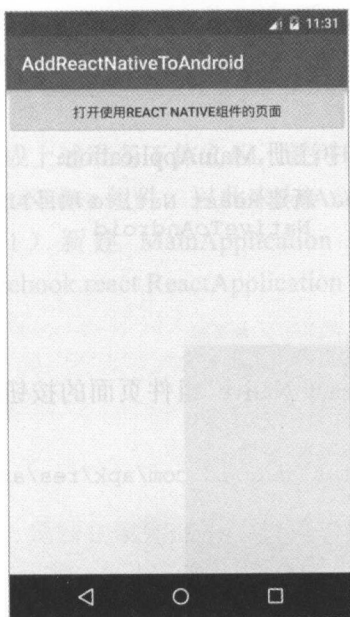


图 9.19 Android 原生工程页面效果

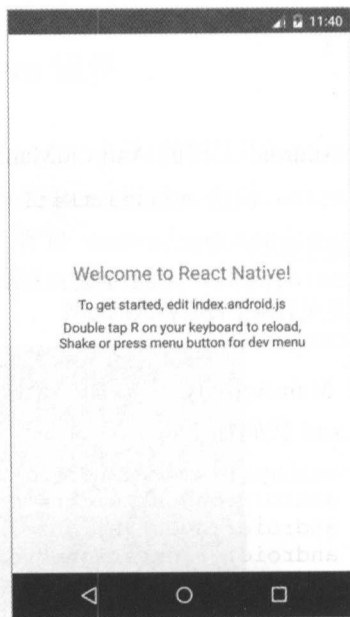



图 9.20 使用 React Native 组件的 Android 原生页面

 **注意：**想要运行 React Native 应用，不要忘记启动 React Native 服务。

9.6 小 结

通过本章的学习，想必读者对基于原生的 React Native 功能扩展有了“矛盾”的感情：

- 一面是原生开发对 React Native 应用具有极大的扩展能力。
- 一面是除了已经掌握的 React Native 开发，还要了解很多原生开发的知识。

其实读者大可放心，因为通常情况下总能找到其他开发者共享的成果。例如，图片选择库的例子就可以使用 `react-native-image-picker` (<https://github.com/marcshilling/react-native-image-picker>)。

 **提示：**想要查找更多第三方组件，可以参考本书第三方组件中介绍的 GitHub (<https://github.com/>) 以及 JS.COACH (<https://js.coach/react-native>)。

本章的作用更多是扩展读者对 React Native 开发的技术视野，了解更多在 React Native 实际开发中解决问题的思路和方法。例如，通过 9.4 节和 9.5 节向已有原生项目添加 React Native 支持的介绍，可以在复用原有原生项目的基础之上，引入 React Native 跨平台开发的优势。

第 10 章 电商 App 的复盘


不知不觉，我们已经使用 React Native 完成了一个电商类应用的开发。在这个过程中，每一章都是一个新的起点，也是一次对 React Native 相关知识由浅入深的积累。

- 第 1 章 为什么要学习 React Native: 介绍了 React Native 的由来、优势，使读者理解为何要学习 React Native，然后介绍了 React Native 基本工具的使用，以及 React Native 项目的基本结构，使读者了解 React Native 开发的基本流程和方法。
- 第 2 章 全局解析 React Native 开发的基础技术: 介绍了 React Native 更多工具的使用，并且着手开发电商类应用的页面，熟悉 React Native 页面开发的方法。
- 第 3 章 React Native 的组件 (1): 深入介绍了 React Native 组件，掌握 React Native 组件和页面开发的更多方法和技巧。
- 第 4 章 React Native 的组件 (2): 介绍了支持特定平台的组件，并分别介绍了 iOS 平台下的使用和 Android 平台下的使用。
- 第 5 章 原生平台的适配和调试: 分别介绍了 iOS 平台和 Android 平台下的适配方法和调试技巧。
- 第 6 章 React Native 的服务器端处理: 介绍了基于 Node.js 的服务器开发，以及 React Native 应用和服务器交互的 fetch API，使读者掌握 React Native 应用网络开发的完整流程。
- 第 7 章 常用 React Native API: 深入介绍了 React Native API，使读者熟练掌握 React Native API，为应用开发更多更强大的功能。
- 第 8~9 章 React Native 与原生平台混合编程: 介绍了原生代码和 React Native JavaScript API 的协同开发。这样不仅可以基于 React Native 平台的组件和 API 进行应用开发，还可以依赖原生平台扩展更多更强大的功能。

从知识结构来看，上述章节已经涉及 React Native 开发的各个方面。而本章将对我们开发的电商 App 做一个完整的回顾和系统的剖析，作为“复盘”，让读者对 React Native 应用有一个架构性的理解。

10.1 电商 App 的文件

以第 8~9 中章的 ch07 工程为例，从 App 的文件组成来回顾电商 App。

提示: ch07 工程包括了完整的功能，如果没有特别说明，本章都是以 ch07 工程为例。

我们开发的 React Native 应用的文件主要有以下 3 个部分:

- JavaScript 文件。

- iOS 原生代码文件。
- Android 原生代码文件。

10.1.1 JavaScript 文件

首先来看电商应用的所有 JavaScript 文件，如图 10.1 所示。



图 10.1 ch07 工程中的 JavaScript 文件

这些 JavaScript 文件的作用如表 10.1 所示。

表 10.1 ch07 工程中JavaScript文件的作用

| 文 件 | 作 用 |
|--|---------|
| app.js, detail.js, more.js | 平台通用的组件 |
| home.ios.js, home.android.js, index.ios.js, index.android.js, main.ios.js, main.android.js | 平台特定的组件 |
| Animation.js | 自定义组件 |
| Communication.js, Deviceinfo.js, ImagePicker.js, Platform.js, Screen.js | 自定义API |

10.1.2 iOS 原生代码文件

iOS 原生代码文件如图 10.2 所示。

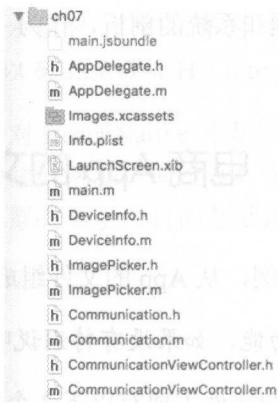


图 10.2 ch07 工程中 iOS 原生代码文件

这些 iOS 原生代码文件的作用如表 10.2 所示。

表 10.2 ch07 工程中iOS原生代码文件的作用

| 文 件 | 作 用 |
|--|----------------------------|
| DeviceInfo.h, DeviceInfo.m, ImagePicker.h, ImagePicker.m, Communication.h, Communication.m | React Native自定义API的iOS原生代码 |
| CommunicationViewController.h, CommunicationViewController.m | iOS页面 |
| AppDelegate.h, AppDelegate.m, Images.xcassets, LaunchScreen.xib, main.m | iOS平台相关文件和资源 |

10.1.3 Android 原生代码文件

Android 原生代码文件如图 10.3 所示。

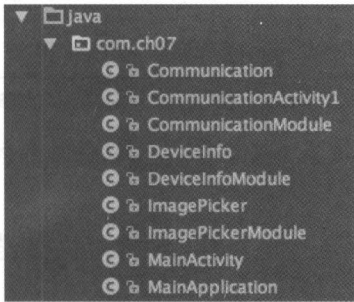


图 10.3 ch07 工程中 Android 原生代码文件

这些 Android 原生代码文件的作用如表 10.3 所示。

表 10.3 ch07 工程中Android原生代码文件的作用

| 文 件 | 作 用 |
|--|--------------------------------|
| Communication.java, CommunicationModule.java, DeviceInfo.java, DeviceInfoModule.java, ImagePicker.java, ImagePickerModule.java | React Native自定义API的Android原生代码 |
| CommunicationActivity1.java, MainActivity.java | Android页面 |
| MainApplication.java | Android平台相关文件 |

10.2 电商 App 的结构

通常应用（也包括这里的 React Native 应用）主要由以下 3 部分构成：

- 页面显示和布局。
- 数据和网络。
- 逻辑控制。

组成结构如图 10.4 所示。

本节就来详细分析应用的这 3 个组成部分。

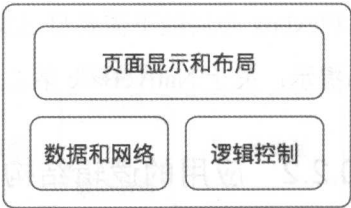


图 10.4 应用的结构

10.2.1 Flexbox 的整体布局

Flexbox 布局是 React Native 应用的基础和重点,因此首先就来梳理电商 App 的 Flexbox 整体布局。以电商 App 的首页为例, Flexbox 的整体布局如图 10.5 所示。

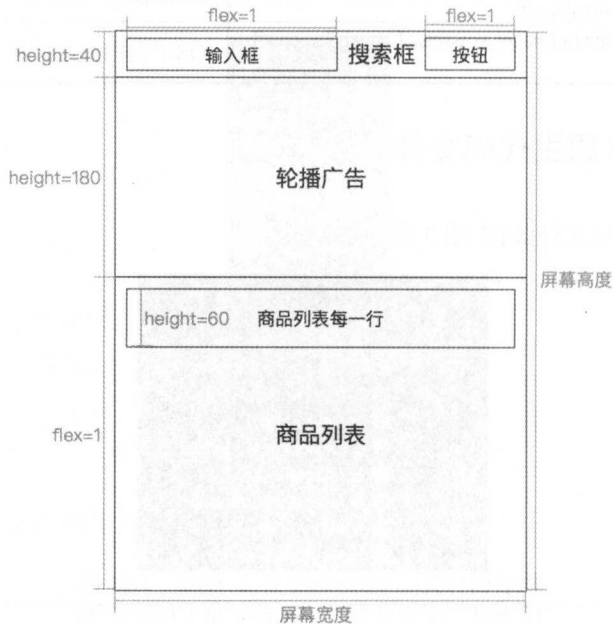


图 10.5 首页的 Flexbox 布局

其中, 商品列表每一行的 Flexbox 布局如图 10.6 所示。

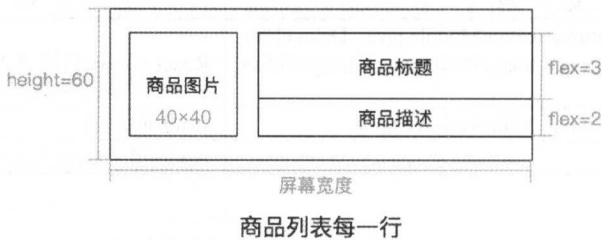



图 10.6 商品列表每一行的 Flexbox 布局

这里需要提醒读者的是, 如果使用如 NativeBase 这样的第三方组件库, 那么常用组件的 Flexbox 布局就不需要自己实现了, 从而可以大大节约设计和开发的成本。

提示: 关于 NativeBase 第三方组件库的详细介绍, 读者可以参考本书 4.2 节。

10.2.2 应用的逻辑结构

在了解了页面和布局之后, 接着来看应用启动时的逻辑, 如图 10.7 所示。

⚠注意：上述逻辑以 iOS 平台为例，其中，TabBarIOS 组件是 iOS 平台的组件，Android 平台的逻辑和 iOS 平台类似，只需要将 TabBarIOS 组件替换成 ViewPagerAndroid 组件即可。

在应用启动之后，页面控制和跳转的逻辑如图 10.8 所示。

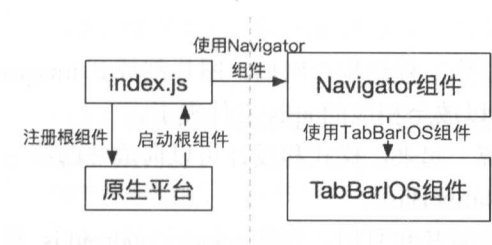


图 10.7 应用启动的逻辑

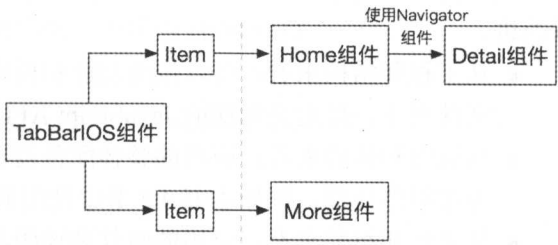


图 10.8 页面控制和跳转的逻辑

其中，Home 组件和 More 组件是通过 TabBarIOS 组件来控制 and 切换的，当单击 Home 组件中的商品列表时，通过 Navigator 组件来实现 Detail 组件的跳转。

⚠提示：关于 Navigator 组件的详细介绍，读者可以参考本书 3.6 节。

10.2.3 应用的通信过程

除了页面布局以及应用逻辑之外，应用还需要处理和服务器的数据交互。在本书 6.4 节中使用了 fetch API 来实现数据交互，如图 10.9 所示。

从服务器获取的数据还可以保存到本地，以提升应用的体验，例如，当网络断开或正在获取数据时，可以将之前保存在本地的数据显示出来，如图 10.10 所示。

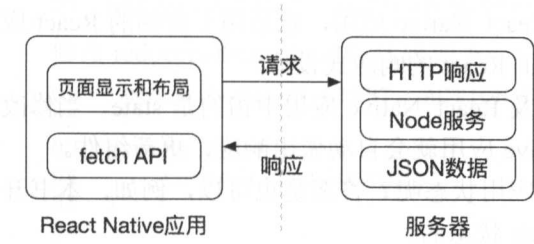


图 10.9 应用的网络通信

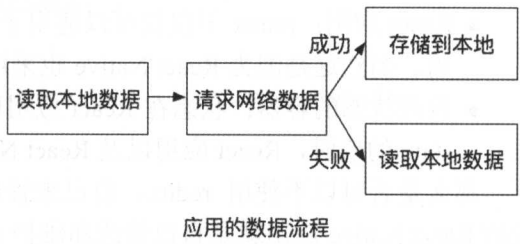


图 10.10 应用的数据流程

除了上述与服务器、数据库的数据交互之外，还有页面间的数据通信，如图 10.11 所示。其中

- 上一级页面传递数据到下一级页面：基于 props 属性（关于 props 的详细介绍，读者可以参考本书 2.6.4 节）。
- 下一级页面返回数据到上一级页面：基于 props 属性和回调函数（关于回调函数的详细介绍，读者可以参考本书 6.5.2 节）。

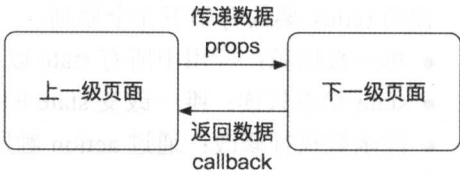


图 10.11 应用的页面间通信

10.3 优化和改进

回顾完应用的整体架构和流程，最后再来审视一下我们的应用，看看还有哪些方面可以改进。

- 从工程和结构来看：可以按照功能和模块划分工程结构。例如，图片都放到 `images` 文件夹下，自定义实现的 JavaScript API 可以放至相应的 `apis` 文件夹下。
- 从页面和体验来看：应用的整体配色需要统一起来，样式和设计可以向成熟的第三方库组件靠拢，例如本书 4.2 节中使用的 `NativeBase`。
- 从文件和组件来看：公用的组件和逻辑尽量封装和复用。例如，`index.android.js` 和 `index.ios.js` 中的大部分组件、样式以及逻辑都是相同的，因此可以考虑复用这部分实现。
- 从网络和数据来看：所有与服务器交互的接口可以单独封装成一个 `network` 的模块，所有与数据库交互的逻辑可以单独封装成一个 `db` 的模块。
- 从整体的架构来看：页面间的数据通信和流程可以使用基于 `redux` (<https://github.com/reactjs/redux>) 的设计。

10.3.1 redux 是什么

这里笔者就以 `redux` 为例，来介绍 React Native 的一种最常用的架构方法来改进 React Native 应用。

简单来说，`redux` 是一个用于管理 React 应用状态的容器。对于这个定义，读者需要注意以下两点。

- **React 应用**：`redux` 不仅仅可以适用于 React Native 应用，也适用于前端的 React 应用。当然这是因为 React Native 也采用了 React 的响应式设计。
- **管理状态的容器**：状态在 React 应用以及 React Native 应用中指的是 `state`，当修改 `state` 的值时，React 应用以及 React Native 应用就会自动做出响应，更新组件。

那么是否可以不使用 `redux`，自己来管理应用状态呢？答案是也可以，例如，本书开发的 React Native 应用就是自己修改和维护 `state` 状态的。

但是在实际开发中，随着应用功能的增加，需要维护的状态也越来越复杂，例如，网络数据、UI 更新、本地数据存储等，这些都是在 React Native 应用中需要处理的，而且这些大多是异步操作。`redux` 就是为了方便开发者解决这些问题，将所有的变化进行统一的流程处理，使应用的状态变化清晰可见。

使用 `redux` 架构有如下 3 个原则。

- **单一数据源**：应用中所有 `state` 以对象树形式存储在一个 `store` 中。
- **state 状态只读**：唯一改变 `state` 的方法是 `action`。
- **纯函数执行修改**：通过 `action` 触发 `reducers` 来改变 `state` 树。

10.3.2 redux 代码示例

也许读者对于上述 redux 的介绍还是很困惑，因此下面将通过 redux 实现修改应用主题的功能来进一步认识 redux 架构。

(1) 使用 React Native 命令行工具来初始化一个新的 ch08 项目。

```
react-native init ch08
```

(2) 将 redux 相关的第三方库添加到 ch08 项目中，安装的命令如下：

```
npm install --save redux
npm install --save react-redux
```

(3) 按照 redux 的 3 个原则分别添加 action 和 reducers。

新建 actions 文件夹并在该文件夹下添加 theme.js 文件，修改 actions/theme.js 代码如下：

```
01 export const ACTION_LIGHT = 'action_light';
02 export const ACTION_DARK = 'action_dark';
03
04 export function setLightTheme() {
05   return {type: ACTION_LIGHT};
06 }
07
08 export function setDarkTheme() {
09   return {type: ACTION_DARK};
10 }
```

新建 reducers 文件夹并在该文件夹下添加 index.js 和 theme.js 文件，修改 reducers/index.js 代码如下：

```
01 import {combineReducers} from 'redux';
02
03 import theme from '../theme';
04
05 export default combineReducers({theme});
```

修改 reducers/theme.js 文件代码如下：

```
01 import {ACTION_LIGHT, ACTION_DARK} from '../actions/theme';
02
03 export type State = {
04   themeState: string
05 }
06
07 const initialState = {
08   themeState: 'light'
09 };
10
11 export default function(state : State = initialState, action) : State {
12   switch(action.type) {
13     case ACTION_LIGHT:
14       return {
15         ...state,
16         themeState: 'light'
17       };
18     case ACTION_DARK:
19       return {
20         ...state,
21         themeState: 'dark'
```

```

22         };
23         default:
24             return state
25     }
26 }

```

添加完 action 和 reducers 之后, 还需要将 redux 应用到 React Native 应用的根组件, 修改 index.ios.js 和 index.android.js 代码如下:

```

01 import React, {Component} from 'react';
02 import {AppRegistry} from 'react-native';
03
04 import {createStore} from 'redux';
05 import {Provider} from 'react-redux';
06
07 import reducers from './reducers';
08 import App from './app';
09
10 class ch08 extends Component {
11     constructor() {
12         super();
13         this.state = {
14             store: createStore(reducers)
15         };
16     }
17
18     render() {
19         return (
20             <Provider store={this.state.store}>
21                 <App></App>
22             </Provider>
23         );
24     }
25 }
26
27 AppRegistry.registerComponent('ch08', () => ch08);

```

(4) 在根组件中引用的 App 组件, 在 app.js 文件中定义, 相应代码如下:

```

01 import React, {Component} from 'react';
02 import {AppRegistry, StyleSheet, Text, View, Button} from 'react-native';
03
04 import {connect} from 'react-redux';
05
06 import reducers from './reducers';
07 import {setLightTheme, setDarkTheme} from './actions/theme';
08
09 class app extends Component {
10     render() {
11         return (
12             <View style={styles.container}>
13                 <Text style={styles.theme}>
14                     当前主题: {this.props.theme.themeState}
15                 </Text>
16                 <Button title='设置 light 主题'
17                     onPress={this.props.onLightThemeClick}></Button>
18                 <Button title='设置 light 主题'
19                     onPress={this.props.onDarkThemeClick}></Button>
20             </View>
21         );
22     }
23 }

```



```

22     }
23   }
24
25   const styles = StyleSheet.create({
26     container: {
27       flex: 1,
28       justifyContent: 'center',
29       alignItems: 'center',
30       backgroundColor: '#F5FCFF'
31     },
32     theme: {
33       fontSize: 20,
34       textAlign: 'center',
35       margin: 10
36     }
37   });
38
39   const mapStateToProps = (state) => {
40     return {theme: state.theme}
41   }
42
43   const mapDispatchToProps = (dispatch) => {
44     return {
45       onLightThemeClick: () => dispatch(setLightTheme()),
46       onDarkThemeClick: () => dispatch(setDarkTheme())
47     }
48   }
49
50   export default connect(mapStateToProps, mapDispatchToProps)(app);

```

上述代码中，通过 `connect()` 方法将 `action` 定义的函数和 `redux store` 中的 `state` 状态映射成了 `App` 组件的属性。

(5) 重新编译和运行应用，效果如图 10.12 所示。

单击“设置 light 主题”或“设置 drak 主题”按钮，就可以改变当前的主题，效果如图 10.13 所示。



图 10.12 基于 `redux` 的应用

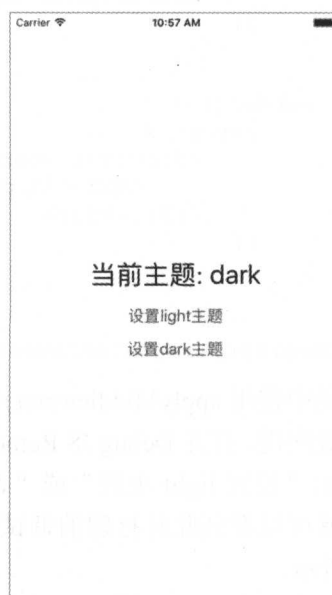


图 10.13 基于 `redux` 的切换主题功能

通常，应用会有多个组件或页面，那么就可以按照 App 组件的写法添加更新主题的逻辑。这样当主题设置改变时，所有组件或页面的主题都会随之更新。

从这个例子可以看出：基于 `redux` 的设计和效率确实优于自己维护 `state` 状态，以及通过 `props` 属性和回调函数实现的页面间数据通信。

10.3.3 redux 生态

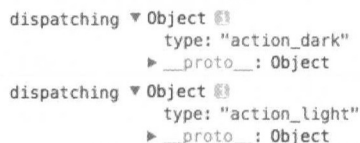
除了基于 `redux` 架构来管理我们应用的 `state` 状态，`redux` 还可以通过使用中间件实现更多强大的功能。例如，当 `state` 发生变化时，可以添加一些打印信息用于调试和跟踪，修改 `index.ios.js` 和 `index.android.js` 代码如下：

```
01 import React, {Component} from 'react';
02 import {AppRegistry} from 'react-native';
03
04 import {createStore, applyMiddleware} from 'redux';
05 import {Provider} from 'react-redux';
06
07 import reducers from './reducers';
08 import App from './app';
09
10 const logger = store => next => action => {
11   console.log('dispatching', action);
12   let result = next(action);
13   return result;
14 }
15 let middlewares = [logger];
16 let createAppStore = applyMiddleware(...middlewares)(createStore);
17
18 class ch08 extends Component {
19   constructor() {
20     super();
21     this.state = {
22       store: createAppStore(reducers)
23     };
24   }
25
26   render() {
27     return (
28       <Provider store={this.state.store}>
29         <App></App>
30       </Provider>
31     );
32   }
33 }
34
35 AppRegistry.registerComponent('ch08', () => ch08);
```

上述代码中使用 `applyMiddleware()` 函数来为创建的 `redux` store 添加 `logger` 中间件。

重新加载应用，打开 `Debug JS Remotely` 调试选项，然后单击“设置 `light` 主题”或“设置 `drak` 主题”按钮，就可以看到此时打印的调试信息，效果如图 10.14 所示。

除了上述我们自己实现的 `logger` 中间件之外，




```
dispatching ▼ Object {
  type: "action_dark"
  __proto__: Object
}
dispatching ▼ Object {
  type: "action_light"
  __proto__: Object
}
```

图 10.14 `redux` 的 `logger` 中间件

还有很多第三方库可以使用。

- `redux-thunk` (<https://github.com/gaearon/redux-thunk>)：一个流行的 `redux` 异步 action 中间件。
- `redux-saga` (<https://github.com/redux-saga/redux-saga>)：用于管理 `redux` 应用异步操作的中间件。
- `redux-persist` (<https://github.com/rt2zz/redux-persist>)：持久化存储 state 的中间件。
- `redux-form` (<https://github.com/erikras/redux-form>)：一个基于 `redux` 的表单。
- `redux-devtools` (<https://github.com/gaearon/redux-devtools>)：用来实时监控 `redux` 的状态 store 的远程调试工具。
- `redux-react-native-i18n` (<https://github.com/derzunov/redux-react-native-i18n>)：基于 `redux` 的国际化解决方案。

从这些第三方库可以看出，`redux` 已经不仅仅是一个第三方库或 `React` 应用架构方案，已经形成了一个基于 `redux` 的完成生态系统，包括组件、异步处理、存储、调试、国际化等一系列功能。

 提示：想要了解 `redux` 生态的更多内容，读者可以参考开源项目 `redux-ecosystem-links` (<https://github.com/mark Erikson/redux-ecosystem-links>)。

10.4 用到的组件

第 3~4 章着重介绍了 `React Native` 组件的使用，分别有：

- `View`;
- `Text`;
- `TextInput`;
- `Button`;
- `ScrollView`;
- `ListView`;
- `Alert`;
- `TouchableHighlight`;
- `StatusBar`;
- `RefreshControl`;
- `Image`;
- `Navigator`;
- `TouchableOpacity`;
- `TabBarIOS/ViewPagerAndroid`;
- `ActivityIndicator`;
- `MapView`;
- `Picker`;
- `Slider`;

- Switch;
- WebView。

但是相比 React Native 庞大的组件库，还有一些本书没有涉及。对于这些组件，大致可以分为以下 3 类。

- 平台特定组件：DataPickerIOS、DrawerLayoutAndroid、NavigatorIOS、PickerIOS、ProgressBarAndroid、ProgressViewIOS、SegmentedControlIOS、ToolbarAndroid。由于这些组件只针对特定平台有效，所以通常情况下，笔者不推荐使用这些组件。
- 细节和体验优化组件：KeyboardAvoidingView。这类组件可以提升用户体验，本书主要介绍组件的基本功能和使用，因此没有详细介绍。
- 可以使用第三方代替组件：Modal。这些组件可以使用简单易用的第三方库代替，例如，弹出框第三方库可以使用 react-native-dialogs（<https://github.com/aakashns/react-native-dialogs>）或 react-native-popup-dialog（<https://github.com/jacklam718/react-native-popup-dialog>）。

虽然上述组件本书没有详细介绍，但是笔者认为读者在掌握本书介绍的 React Native 知识之后，完全有能力自己学习和使用这些组件。

10.5 小 结

本章对我们开发的 React Native 应用做了一个完整的总结和回顾，包括：

- 项目的文件和结构。
- 页面的整体布局。
- 应用的逻辑。
- 数据通信和本地存储。

同时，本章还介绍了 redux 架构及其生态，为复杂 React Native 应用的设计和架构提供了思路。

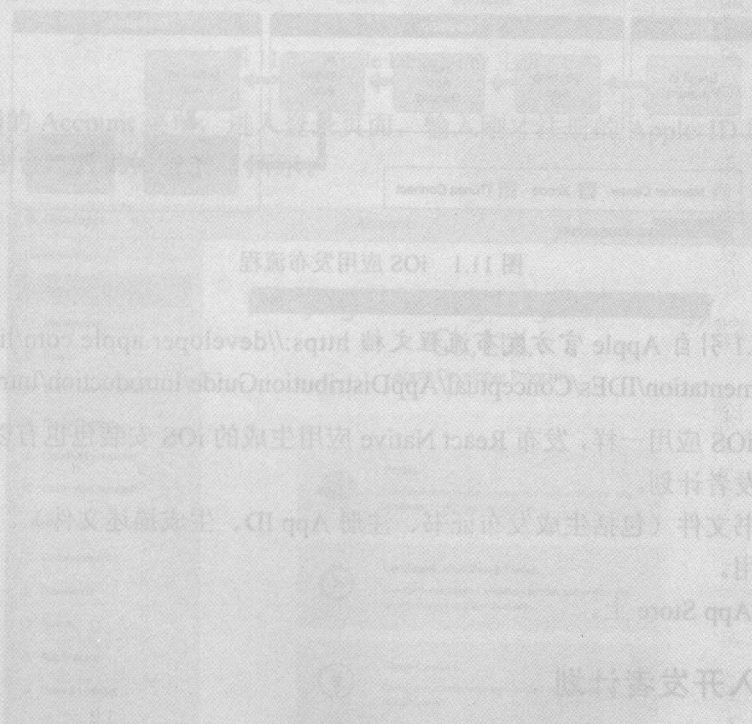
至此，React Native 应用开发阶段的所有内容都已经呈现在读者的面前。

第 4 篇

App 的发布和更新

第 11 章 App 的发布

第 12 章 App 的热部署



第 11 章 App 的发布

当完成 React Native 应用的开发工作之后，就需要将应用打包发布到应用商店。

- iOS 应用商店：App Store (<http://www.apple.com/cn/itunes/charts/>)。
- Android 应用商店：Google Play (<https://play.google.com/store>)、腾讯应用宝 (<http://sj.qq.com/>)、百度手机助手 (<http://shouji.baidu.com/>)、360 手机助手 (<http://sj.360.cn/>)、小米商店 (<http://app.mi.com/>) 等。

通过打包发布，最终让用户能够在自己的手机上下载、安装和使用我们的 App。本章主要内容就是以上两部分。

11.1 App Store 苹果应用商店

关于如何发布 iOS 应用到 App Store，苹果官方有详细的介绍和说明，文档地址如下：
<https://developer.apple.com/support/cn/>

如图 11.1 描述了 iOS 应用发布的主要流程。

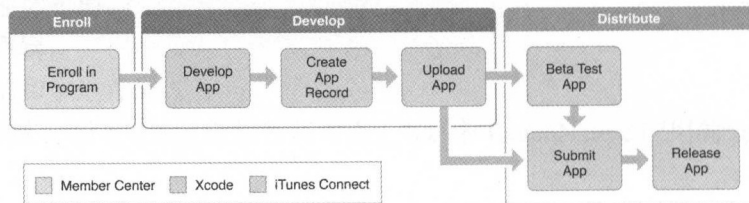


图 11.1 iOS 应用发布流程

 提示：图 11.1 引自 Apple 官方发布流程文档 <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/Introduction/Introduction.html>。

和普通的 iOS 应用一样，发布 React Native 应用生成的 iOS 安装包也有以下几个步骤。

- 加入开发者计划。
- 生成证书文件（包括生成发布证书、注册 App ID、生成描述文件）。
- 打包应用。
- 发布到 App Store 上。

11.1.1 加入开发者计划

加入开发者计划主要有两个条件：Apple ID 以及缴纳费用。

1. 注册Apple ID

首先我们需要一个 Apple ID，打开 Apple ID 的注册地址 <https://appleid.apple.com/account#!&page=create>，效果如图 11.2 所示。

图 11.2 注册 Apple ID

然后按照网站的说明和要求，填写相关信息完成注册。成功注册 Apple ID 之后，打开 Apple Developer（<https://developer.apple.com/>），效果如图 11.3 所示。



图 11.3 Apple Developer 主页

单击右侧的 Account 菜单，进入登录页面，输入刚才注册的 Apple ID 即可进入 Apple Developer 控制台，效果如图 11.4 所示。

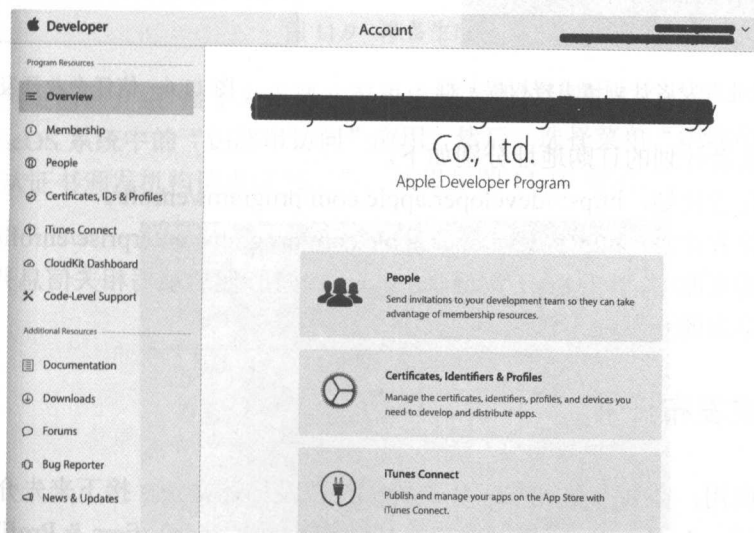



图 11.4 Apple Developer 控制台

2. 申请加入苹果开发者计划

苹果提供了以下两种开发者计划。

- 个人开发者计划：每年费用是\$99，应用可以发布至 App Store。
- 企业开发者计划：每年费用是\$299，应用可以不用发布至 App Store。

 **提醒：**如果是企业开发者计划，用户下载并安装后，打开应用会弹出请求授权的提示框，效果如图 11.5 所示。

此时，用户需要打开“设置”下的“通用”→“描述文件与设备管理”，选择刚才安装的应用，单击信任按钮后才能打开应用，效果如图 11.6 所示。



图 11.5 企业开发者计划请求授权提示框

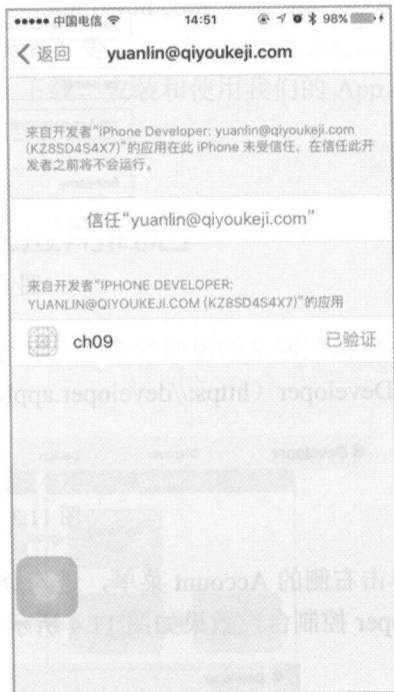


图 11.6 信任企业开发者

这两种开发者计划的订购地址分别如下。

- 个人开发者计划：<https://developer.apple.com/programs/enroll/>。
- 企业开发者计划：<https://developer.apple.com/programs/enterprise/enroll/>。

在上述订购页面中，单击 Start Your Enrollment 按钮，然后填写相关信息并缴纳费用后，就成功加入了苹果的开发者计划。

11.1.2 生成发布证书

打包 iOS 应用，需要准备发布证书、App ID 以及描述文件，接下来先介绍发布证书。

(1) 打开 Apple Developer 控制台，进入 Certificates, Identifiers & Profiles，单击右侧的加号按钮新建发布证书，效果如图 11.7 所示。



图 11.7 新建发布证书

(2) 在证书类型页面选择 App Store and Ad Hoc 选项，效果如图 11.8 所示。

Production

• App Store and Ad Hoc

Sign your iOS app for submission to the App Store or for Ad Hoc distribution.

图 11.8 选择证书类型为 App Store and Ad Hoc

(3) 单击 Continue 按钮，进入生成证书页面，效果如图 11.9 所示。

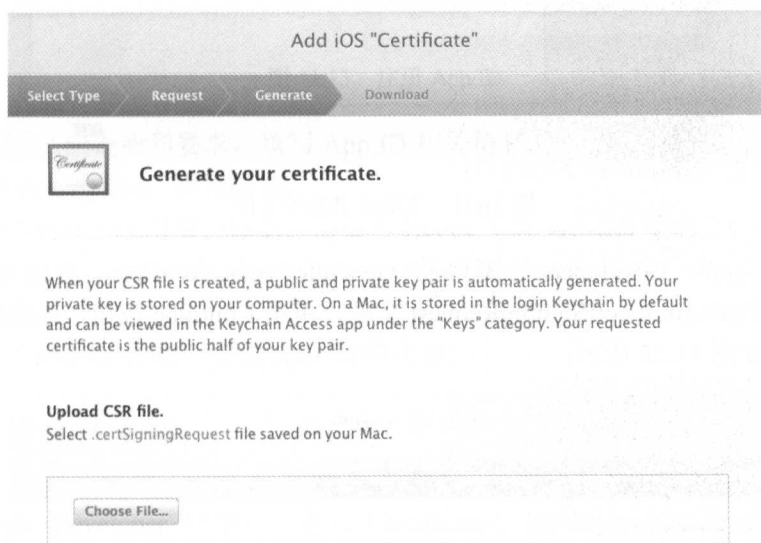


图 11.9 准备生成证书

(4) 按照页面的介绍，接下来需要在苹果电脑上通过钥匙串来生成一个证书签名文件。打开 macOS 系统中的“钥匙串访问”应用，然后，选择菜单“钥匙串访问”→“证书助理”→“从证书颁发机构请求证书...”，效果如图 11.10 所示。

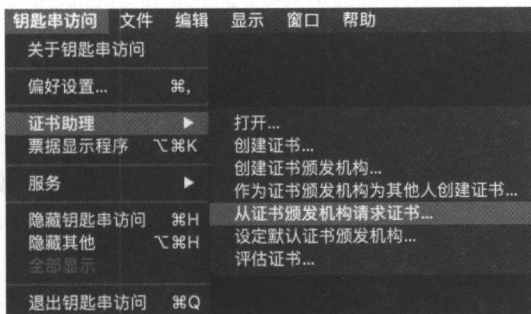


图 11.10 使用“钥匙串访问”应用生成证书签名文件

(5) 填写“用户电子邮件地址”等信息，选择“存储到磁盘”选项，单击“继续”按钮，生成一个默认名为 CertificateSigningRequest.certSigningRequest 的文件并保存至桌面，效果如图 11.11 所示。

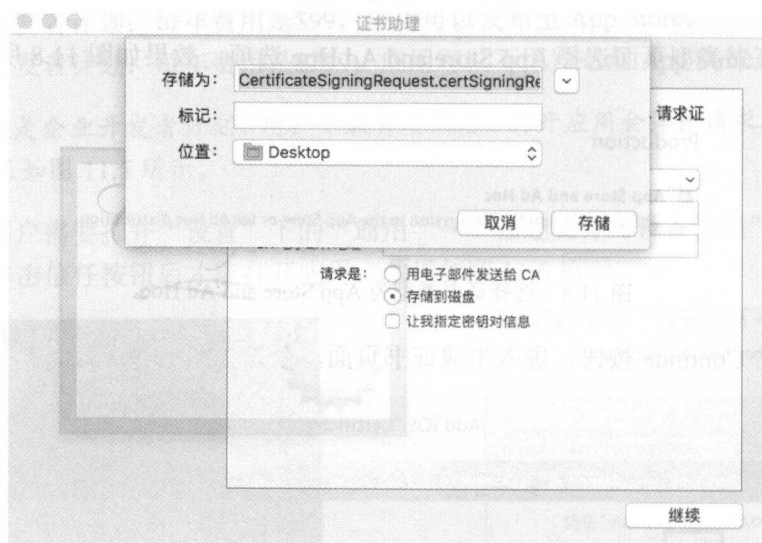


图 11.11 生成证书签名文件

(6) 在 Apple Developer 控制台的 Generate your certificate 页面选择磁盘上的 CertificateSigningRequest.certSigningRequest 文件，单击 Continue 按钮，就完成了发布证书的创建，效果如图 11.12 所示。

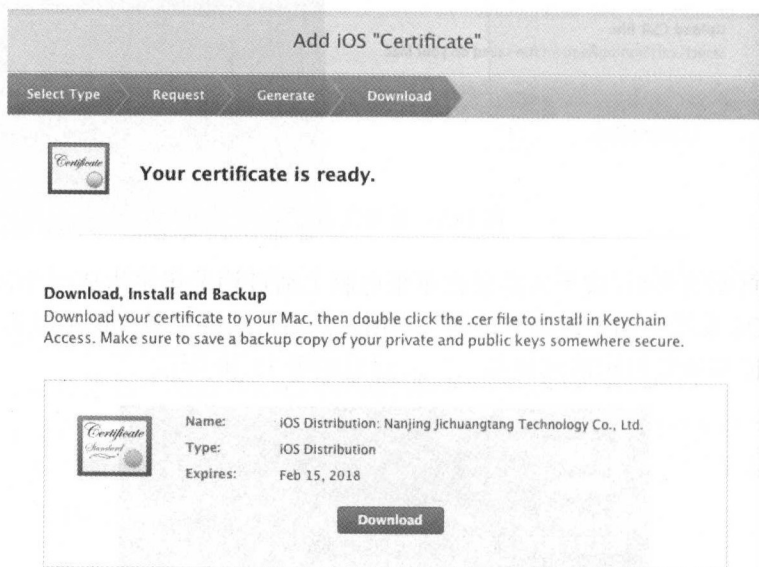


图 11.12 生成发布证书

(7) 成功生成发布证书后，单击 Download 按钮下载该证书。最后双击下载的证书文件，完成证书的安装。

11.1.3 注册 App ID

打开 Apple Developer 控制台，进入 Certificates, Identifiers & Profiles，选择左侧 Identifiers 一栏中的 App IDs，单击右侧的加号按钮，效果如图 11.13 所示。

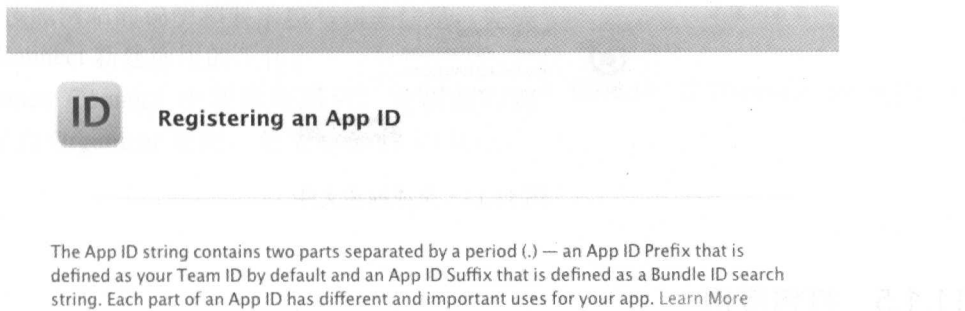


图 11.13 注册 App ID

然后按照网站的说明和要求，填写 App ID 相关信息。

- App ID Description: 应用描述。
- App ID Suffix: 应用唯一标识，需要与 Xcode 工程 General 中的 Bundle Identifier 保持一致。当然，还可以使用 Wildcard App ID 的通配符 “*” 来匹配多个 Bundle Identifier。

最后单击 Continue 按钮，完成 App ID 的注册。

11.1.4 生成描述文件

打开 Apple Developer 控制台，进入 Certificates, Identifiers & Profiles，选择左侧 Provisioning Profiles 一栏中的 Distribution，单击右侧的加号按钮新建描述文件。

然后在描述文件类型页面选择 App Store，效果如图 11.14 所示。



图 11.14 选择描述文件类型为 App Store

接着选择刚才注册的 App ID 和新建的发布证书，单击 Continue 按钮，输入 Profile Name，即可生成描述文件，效果如图 11.15 所示。


成功生成描述文件后，单击 Download 按钮下载该文件。最后双击下载的描述文件，完成安装。




图 11.15 生成描述文件

11.1.5 打包应用

准备好发布证书、App ID 以及描述文件之后，就可以开始打包应用了。

 **提示：**本章打包用到的 React Native 应用使用 `react-native init ch09` 命令生成，关于更多 React Native 命令行工具的使用，读者可以参考本书第 3 章的内容。

- (1) 将 Xcode 工程 General 中 Provisioning Profile 设置成 11.1.2 节中生成的描述文件。
- (2) 打开 Xcode 菜单 Product，选择 Archive 开始打包。

 **提示：**如果菜单 Archive 是灰色的话，请检查当前 Xcode 使用的是否为 iPhone 真机。

- (3) 打包成功后，会自动打开菜单 Window→Organizer 的页面，效果如图 11.16 所示。

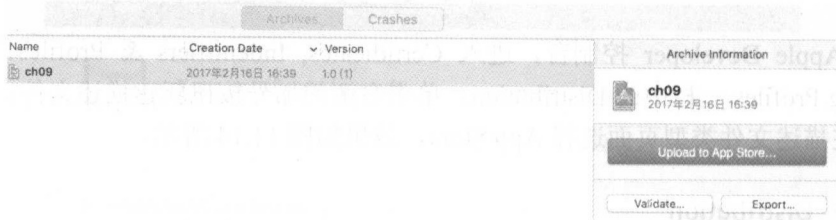


图 11.16 打包成功后自动打开 Organizer 页面

11.1.6 发布到 App Store

完成应用打包工作之后，最后一步就是将应用发布到 iTunes Connect 了。

问题：什么是 iTunes Connect?

回答：iTunes Connect 是一套基于 Web 的工具，可便于开发者提交和管理 App，以供在 App Store 或 Mac App Store 中销售。更多详细介绍，可以参考 https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide_SCh/Chapters/About.html。

(1) 打开 iTunes Connect (<https://itunesconnect.apple.com/>), 选择“我的 App”, 单击左侧的加号按钮, 效果如图 11.17 所示。

(2) 选择“新建 App”选项, 弹出填写 App 信息提示框, 效果如图 11.18 所示。

(3) 按照网站的说明和要求填写相关信息, 完成在 iTunes Connect 新建应用的工作。

在 iTunes Connect 中新建应用后, 重新返回到 Xcode 工程的 Organizer 页面, 效果如图 11.19 所示。

图 11.17 在 iTunes Connect 新建应用



新建 App

平台 ?

☐ iOS ☐ Apple TVOS

名称 ?

主要语言 ?

选取

套装 ID ?

选择

请前往 开发者门户网站 (Developer Portal) 注册一个新的套装 ID。

SKU ?

取消 创建



ch09

2017年2月16日 16:39

Upload to App Store...

Validate...

Export...

Details

Version 1.0 (1)

Identifier org.reactjs.native.example....

Type iOS App Archive

Download dSYMs...

图 11.18 填写发布应用的相关信息

图 11.19 显示打包结果的 Organizer 页面

(4) 单击 Upload to App Store 按钮, 选择相应证书后, 再单击 Upload 按钮, 即可发布应用到 iTunes Connect 中。

提示: 由于网络原因, 可能上传应用的时间较长, 请耐心等待或提高网络带宽。

11.2 Android 应用商店

由于网络原因导致 Google 相关服务在国内无法访问, 因此, 如果只是在国内销售的 Android 应用, 通常不需要上传到 Google Play 商店, 下面介绍的内容也以国内应用商店为主。

11.2.1 生成签名文件

Android 要求所有应用都有一个数字签名才会被允许安装在用户手机上, 因此在把应用发布到应用商店之前, 需要对生成的 APK 包进行签名。

提示: 关于 Android 签名和发布的详细介绍, 读者可以参考 Android 官方文档 <https://developer.android.com/studio/publish/app-signing.html>。

(1) 打开 Android Studio 的菜单 Build→Generate Signed APK..., 效果如图 11.20 所示。

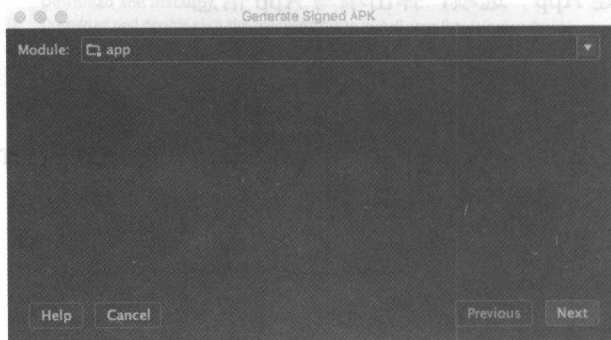


图 11.20 生成签名 APK 包: 配置模块

(2) 单击 Next 按钮, 进入下一页面, 效果如图 11.21 所示。

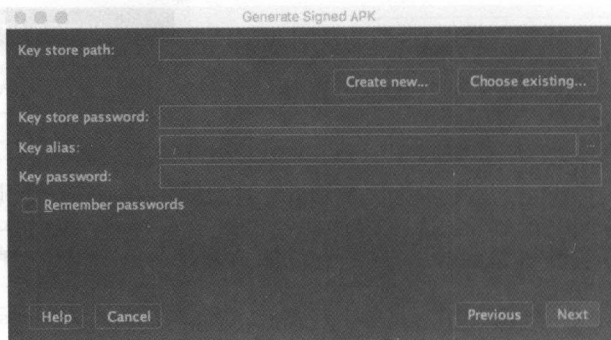


图 11.21 生成签名 APK 包: 配置签名文件

(3) 单击 Create new...按钮, 进入新建签名文件的页面, 效果如图 11.22 所示。

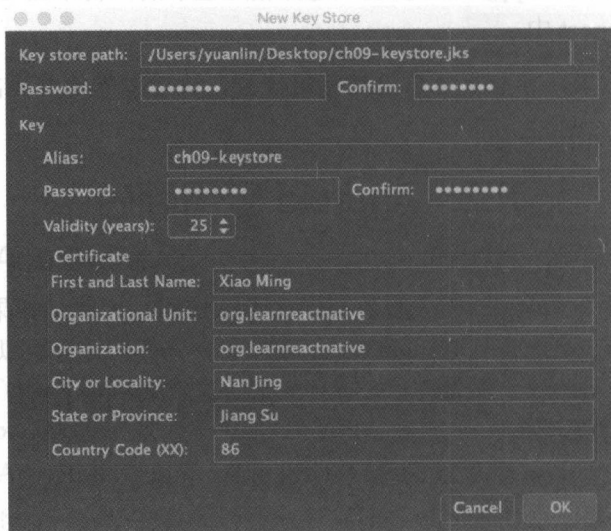



图 11.22 生成签名 APK 包: 新建签名文件

(4) 根据页面的说明和要求, 填写以下相关信息。

- Key store path: 密钥库存放的路径。
- Password: 密钥库密码。
- Key Alias: 签名文件别名。
- Key Password: 签名文件密码。
- Key Validity: 签名文件有效期, 单位为年。
- 名字、组织、城市、省份、国家码等信息。

(5) 单击 OK 按钮, 生成打包 APK 时要使用的签名文件。

 注意: 打包 APK 时会用到上述设置的别名、密码等信息, 所以请读者牢记。

11.2.2 打包应用

和 Android 原生应用打包的过程相比, React Native 应用打包前需要先打包 JavaScript 资源。

1. 打包JavaScript资源

进入 React Native 项目所在的目录:

```
cd ch09
```

新建 assets 文件夹, 用于存放 JavaScript 打包后的资源。

```
mkdir -p android/app/src/main/assets
```

使用 React Native 命令行工具打包 JavaScript 相关资源。

```
react-native bundle --platform android --dev false --entry-file index.
android.js --bundle-output android/app/src/main/assets/index.android.
bundle --assets-dest android/app/src/main/res/
```

打包成功后, ch09 工程的效果如图 11.23 所示。

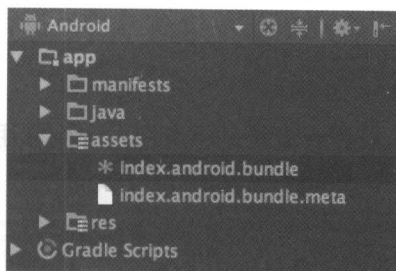


图 11.23 打包 JavaScript 资源

2. 打包应用

准备好签名文件以及 JavaScript 资源包之后, 就可以开始打包应用了。

首先打开 Android Studio 的菜单 Build→Generate Signed APK..., 单击 Next 按钮, 进入配置签名文件页面, 效果如图 11.24 所示。

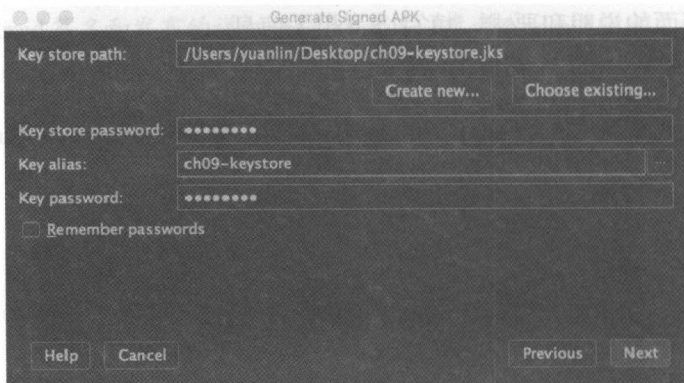


图 11.24 生成签名 APK 包：配置签名文件

选择刚才生成的签名文件，并填写别名、密码等信息。然后单击 Next 按钮，进入配置编译选项页面，效果如图 11.25 所示，配置生成 APK 包存放的路径以及编译类型为 release。此时单击 Finish 按钮，Android Studio 就会在指定目录生成 APK 包。

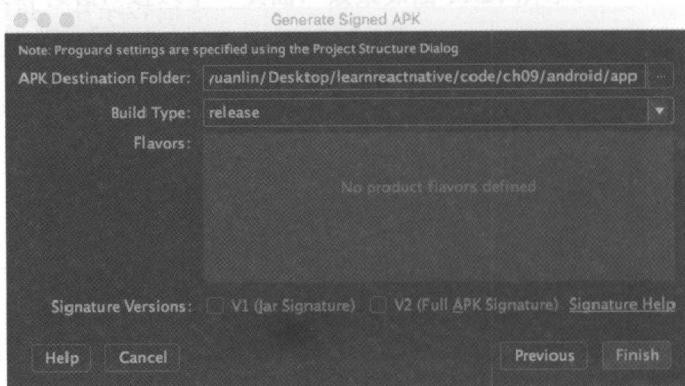


图 11.25 生成签名 APK 包：配置编译选项

11.2.3 发布到应用商店

完成应用打包工作之后，最后一步就是将应用发布到应用商店。众所周知，除了官方的 Google Play 商店，国内的 Android 应用商店也有很多：

- 腾讯应用宝 (<http://sj.qq.com/>)。
- 百度手机助手 (<http://shouji.baidu.com/>)。
- 360 手机助手 (<http://sj.360.cn/>)。
- 华为应用市场 (<http://app.hicloud.com/>)。
- OPPO 软件商店 (<http://store.oppomobile.com/>)。
- 小米商店 (<http://app.mi.com/>)。
- 安卓市场 (<http://apk.hiapk.com/>)。
- 安智市场 (<http://www.anzhi.com/>)。
- 豌豆荚 (<https://www.wandoujia.com/>)。

以及很多还没有列出的其他应用商店。


通常的办法就是去每个应用商店注册应用信息，然后将我们的应用发布到每一个应用商店。这个办法虽然可行但是并不推荐，因为效率实在太低。

这里笔者推荐一个一站式发布的平台酷传(<http://www.kuchuan.com/>)，其功能如图 11.26 所示。



图 11.26 酷传

酷传最大的作用是支持一键发布 30 多个应用市场，从而大大节省了 Android 应用发布的时间和精力。关于酷传的详细使用，读者可以自行注册酷传账号，按照页面说明和要求完成操作即可。

 **提示：**解决发布 Android 应用到多应用商店的问题，本书推荐但不局限于酷传。

11.3 小 结

本章介绍了一个 App 完成后要做的工作，App 如果发布，并不像普通软件一样，给个 exe 文件就能运行，要让别人能看到我们的应用，就必须遵守苹果或 Android 的规定，比如 11.1 节的内容，就是帮助我们完成 App 在苹果商店的上架，11.2 节的内容就是帮助我们完成 App 在 Android 一些商店的上架，尤其是最后，笔者还推荐了一款可解决应用同时上架多个商店的软件。


第 12 章 App 的热部署

第 11 章详细介绍了 iOS 应用以及 Android 应用打包和发布的全过程。其中有一个问题容易被忽略：应用发布到应用商店是需要审核的。通常，Android 应用商店审核大概需要几天，iOS 应用商店审核却比较长，大概需要 1~2 周。但是移动应用产品更新的速度却非常快，通常 1~2 周或者几天时间就可完成一次版本迭代。

热部署就是为了解决这一问题而生的，不用提交新的审核，通过服务器动态更新 React Native 的 JavaScript 代码来实现应用的更新。

12.1 什么是热部署

在详细解释热部署之前，我们可以想一想平时使用到的例子。通常的计算机上都会安装 Office 办公软件，如果 Office 有新功能新版本推出，则需要下载更新的安装包到计算机上，然后安装后使用。人们浏览的新闻网站虽然没有下载和更新过任何软件，但是每天访问时的内容都有可能不同，因为网站的内容都是通过浏览器实时从服务器获取的，网站服务器的内容一更新，人们打开的网站页面的内容也会随之更新。

 **提示：**热部署与上述浏览器打开网站的例子并不完全相同，但是通过这个生活中的例子，可以方便读者理解热部署的原理和作用。

对于应用来说，热部署是指不需要重新审核和安装新版本的应用，通过服务器的动态更新来更新应用的功能。热部署的原理如图 12.1 所示。

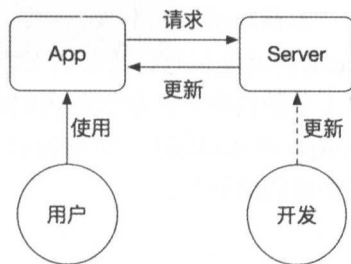


图 12.1 热部署的原理

12.2 解析 React Native 应用的工作原理

React Native 的优势在本书第 1 章中就已经有了详细的介绍。除了之前介绍的跨平台开发优势之外，React Native 的另一项革命性的创新就在于为移动应用的动态更新提供了基础。这里回顾一下 iOS 应用加载 React Native 组件的过程。

(1) 使用 React Native 命令行工具来初始化一个新的项目：

```
react-native init ch010
```

(2) 等待工程创建成功并安装好所有依赖后, 使用 Xcode 打开 `ios/ch12.xcodeproj` 文件, 修改 `AppDelegate` 代码如下:

```

01 #import "AppDelegate.h"
02
03 #import <React/RCTBundleURLProvider.h>
04 #import <React/RCTRootView.h>
05
06 @implementation AppDelegate
07
08 - (BOOL)application:(UIApplication *)application didFinishLaunching
    WithOptions:(NSDictionary *)launchOptions {
09     NSURL *jsCodeLocation;
10
11     jsCodeLocation = [NSURL URLWithString:@"http://localhost:8081/index.
        ios.bundle?platform=ios&dev=true"];
12
13     RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:
        jsCodeLocation
                                moduleName:@"ch10"
                                initialProperties:nil
                                launchOptions:launchOptions];
14     rootView.backgroundColor = [[UIColor alloc] initWithRed:1.0f green:
        1.0f blue:1.0f alpha:1];
15
16     self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].
        bounds];
17     UIViewController *rootViewController = [UIViewController new];
18     rootViewController.view = rootView;
19     self.window.rootViewController = rootViewController;
20     [self.window makeKeyAndVisible];
21     return YES;
22 }
23
24 @end


```

此时 `jsCodeLocation` 加载的地址是 `http://localhost:8081/index.ios.bundle`, 那么这个地址是从何而来呢?

原来, 当执行 `react-native run-ios` 或者 `react-native run-android` 命令时:

- 首先, 将 JavaScript 相关资源打包, 例如, iOS 应用的 JavaScript 资源打包为 `index.ios.bundle`。
- 然后, 启动一个基于 Node.js 的 React Native 服务, 这个服务运行在本地, 监听的默认端口号是 8081, 最后当应用请求该地址的 `index.ios.bundle` 资源时, 就会从 React Native 服务实时获取最新的 JavaScript 资源。
- 最后, 当应用请求该地址的 `index.ios.bundle` 资源时, 就会从 React Native 服务实时获取最新的 JavaScript 资源。
- 另外, 当修改 `index.ios.js` 文件时, React Native 会重新打包 JavaScript 资源, 这样就实现了应用的热更新。

正是 React Native 的这种设计, 让 React Native 平台很容易就可实现其他原生开发平台无法实时更新的优势。

 **小知识:** 原生开发平台通过第三方的解决方案也可以具有热更新的能力, 例如, iOS 平台下的 JSPatch (<http://jspatch.com/>), Android 平台下的 Bugly (<https://bugly.com>)。

qq.com/v2/upgrade)。但是这些都不是官方的解决方案，而且苹果公司已对此发出了警告。因此相比 React Native 这种天生支持热更新的设计来说，它们的普及度和稳定性都有一定差距。

12.3 实现 React Native 的热部署


了解了 React Native 应用的运行原理之后，就可以实现自己的热部署系统了。

12.3.1 服务端实现

服务端实现的步骤如下。

(1) 新建一个用于热更新的服务器项目，命令如下：

```
express --ejs HotUpdate // 新建 HotUpdate 项目，并使用 ejs 模板引擎
```

 提示：关于 express 命令的详细介绍，读者可以参考本书 6.3 节。

(2) 将 React Native 应用用到的 JavaScript 资源打包：

```
react-native bundle --platform ios --dev false
--entry-file index.ios.js --bundle-output ios/
index.ios.bundle
```

(3) 将打包好的 JavaScript 资源包放至 HotUpdate 的 public 目录下，效果如图 12.2 所示。

(4) 运行 HotUpdate 服务，命令如下：

```
cd HotUpdate
npm start
```

(5) 此时，使用浏览器打开地址 <http://localhost:3000/index.ios.bundle>，下载 index.ios.bundle 文件，表示服务器程序已经部署成功。

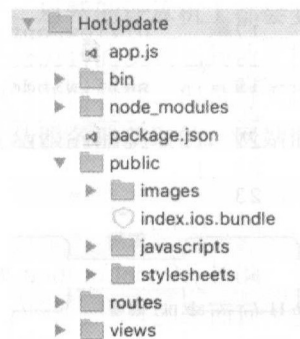


图 12.2 将 JavaScript 资源包放至服务器

12.3.2 客户端实现

服务器部署成功后，客户端的实现就变得简单很多。

(1) 使用 Xcode 打开 ios/ch12.xcodeproj 文件，修改 AppDelegate 代码如下：

```
01 #import "AppDelegate.h"
02
03 #import <React/RCTBundleURLProvider.h>
04 #import <React/RCTRootView.h>
05
06 @implementation AppDelegate
07
08 - (BOOL)application:(UIApplication *)application didFinishLaunching
    WithOptions:(NSDictionary *)launchOptions {
09     NSURL *jsCodeLocation;
```

```

10
11   jsCodeLocation = [NSURL URLWithString:@"http://localhots:3000/
    index.ios.bundle?platform=ios&dev=false"];
12
13   RCTRootView *rootView = [[RCTRootView alloc] initWithBundleURL:
    jsCodeLocation
                                moduleName:@"ch10"
                                initialProperties:nil
                                launchOptions:launchOptions];
14   rootView.backgroundColor = [[UIColor alloc] initWithRed:1.0f green:
    1.0f blue:1.0f alpha:1];
15
16   self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].
    bounds];
17   UIViewController *rootViewController = [UIViewController new];
18   rootViewController.view = rootView;
19   self.window.rootViewController = rootViewController;
20   [self.window makeKeyAndVisible];
21   return YES;
22 }
23
24 @end

```

(2) 此时 `jsCodeLocation` 加载的地址是 `http://localhost:3000/index.ios.bundle`，这个地址就是 12.3.1 节搭建的 HotUpdate 服务器 JavaScript 资源的地址。

(3) 重新编译和运行 iOS 应用，可以看到应用成功加载 React Native 组件的效果，如图 12.3 所示。



图 12.3 加载 HotUpdate 服务器的 JavaScript 资源

(4) 为了验证热更新的效果，此时可以修改 `index.ios.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class ch10 extends Component {

```

```

04     render() {
05         return (
06             <View style={styles.container}>
07                 <Text style={styles.welcome}>
08                     来自 HotUpdate 服务器的 Bundle
09                 </Text>
10                 <Text style={styles.instructions}>
11                     To get started, edit index.ios.js
12                 </Text>
13                 <Text style={styles.instructions}>
14                     Press Cmd+R to reload,{'\n'}
15                     Cmd+D or shake for dev menu
16                 </Text>
17             </View>
18         );
19     }
20 }
21
22 // 这里省略了没有修改的代码

```

(5) 重新打包 React Native 应用的 JavaScript 资源:

```

react-native bundle --platform ios --dev false --entry-file index.ios.js
--bundle-output ios/index.ios.bundle

```

(6) 将生成的新的 JavaScript 资源包覆盖 HotUpdate 工程相应的文件。重新加载应用，可以看到 React Native 应用在未做任何修改的情况下，页面内容已经成功更新，效果如图 12.4 所示。



图 12.4 React Native 应用热更新

至此，一个简单的热更新服务就实现了。

12.4 微软的热部署方案 CodePush

我们自己实现的热更新服务虽然简单有效，但是也存在一些问题：

- 首次使用有一定延时，因为需要下载 JavaScript 包。
- 网络劫持等问题。

因此，实际开发中使用的热更新通常基于本地文件，即将更新包下载到本地后使用。基于上述原理，可以使用第三方的实现方案，即微软推出的 CodePush。

12.4.1 CodePush 简介

CodePush (<https://microsoft.github.io/code-push/>) 是微软提供的一套用于热更新 React Native 和 Apache Cordova 应用的服务，提供 React Native 和 Apache Cordova 开发者直接部署移动应用更新给用户的云服务。CodePush 作为一个中央仓库，开发者可以推送更新，然后应用从客户端查询更新。这样不需要重新审核和安装应用，就可以解决缺陷和添加新的特性。

CodePush 主要的功能和优势有：

- 开源免费，CodePush 源码托管在 GitHub (<https://github.com/microsoft/code-push>)。
- 直接对用户部署代码更新。
- 管理 Alpha、Beta 以及生产环境应用。
- 支持 JavaScript 文件和图片资源的更新。
- 支持 React Native 和 Apache Cordova 应用。

12.4.2 CodePush 安装和注册

使用 CodePush 之前首先要安装 CodePush 命令行工具，然后新建 CodePush 账号和应用。

1. 安装CodePush命令行工具

使用 CodePush 都是通过 CodePush 官方提供的命令行工具，安装命令如下：

```
npm install -g code-push-cli
```

安装成功后可以通过如图 12.5 所示的方法进行验证。

```
+ ~ code-push -v
1.12.6-beta
+ ~
```

2. 新建CodePush账号

在终端输入命令 `code-push register`，将会打开注册页面让开发者选择授权账号，效果如图 12.6 所示。

图 12.5 查看是否安装成功并显示版本号

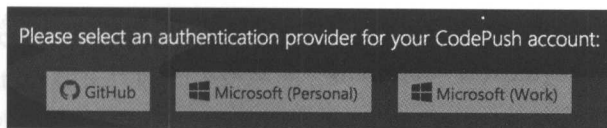


图 12.6 新建 CodePush 账号

选择相应的账号登录并授权，会得到一个 CodePush“access key”，复制此 access key 到终端即可完成注册，效果如图 12.7 所示。

```
A browser is being launched to authenticate your account. Follow the instructions it displays to complete your login.

Enter your access key: AkHF0-6fxAs8IzsZAt8VBV-0ftwm4krrevS6b

Successfully logged-in. Your session file was written to /Users/yuanlin/.code-push.config. You can run the code-push logout command at any time to delete this file and terminate your session.
```


图 12.7 登录 CodePush 账号

用于账号和登录管理的命令还有以下几项。

- code-push login: 登录。
- code-push logout: 注销。
- code-push access-key ls: 列出 access-key。
- code-push access-key rm <accessKey>: 删除某个 access-key。

3. 新建CodePush应用

为了让 CodePush 服务器知道我们的应用，还需要在服务器上进行注册，在终端输入命令 code-push app add <appName> 即可完成注册，效果如图 12.8 所示。

提示：如果 React Native 应用同时发布 Android 和 iOS 版本，那么在注册 CodePush 应用的时候需要注册两个应用，获取两套 deployment key，例如 code-push app add App-Android 和 code-push app add App-iOS。

```
* ~ code-push app add CodePush-iOS
Successfully added the "CodePush-iOS" app, along with the following default deployments:

+-----+-----+
| Name   | Deployment Key |
+-----+-----+
| Production | WCXLKgBm0VP22joECJQVomPFijjD4krrevS6b |
+-----+-----+
| Staging   | 0RenElCPn5l4UTFggU4zlpS6Qm4krrevS6b |
+-----+-----+


* ~ code-push app add CodePush-Android
Successfully added the "CodePush-Android" app, along with the following default deployments:

+-----+-----+
| Name   | Deployment Key |
+-----+-----+
| Production | gz16pfadUMLdMd-YYjcRmIXg8uEW4krrevS6b |
+-----+-----+
| Staging   | ku0J5ndA3s_D7hHuW0WeG60FV5Jl4krrevS6b |
+-----+-----+
```

图 12.8 新建 CodePush 应用

成功新建 CodePush 应用后，每个应用都会得到两个 deployment key。

- Production: 用于生产环境的 deployment key。
- Staging: 用于模拟环境的 deployment key。

提示：上述应用的 deployment key 在后面的配置中需要用到。当然，还可以使用命令 code-push deployment ls <appName> -k 查询。

用于应用管理的命令还有以下几项。

- code-push app add: 在登录账号中添加一个新的 App。

- code-push app remove: 在登录账号中删除一个 App。
- code-push app rename: 重命名一个存在的 App。
- code-push app list: 列出登录账号下面所有的 App。
- code-push app transfer: 把 App 的所有权转移到另外一个账号。

12.4.3 集成 CodePush SDK

完成了上述 CodePush 账号和应用操作之后，下一步就可以将 CodePush SDK 集成到 React Native 的应用中，实现热更新了。

(1) 使用 React Native 命令行工具来初始化一个新的项目。

```
react-native init CodePush
```

(2) 将 react-native-code-push 添加到新建的 CodePush 项目中，命令如下：

```
cd CodePush
npm install --save react-native-code-push@latest
```

(3) 将 react-native-code-push 的依赖添加到原生工程中，命令如下：

```
react-native link react-native-code-push
```

(4) 此时会提示分别输入 iOS 和 Android 应用的 deployment key:

```
? What is your CodePush deployment key for iOS (hit <ENTER> to ignore)
? What is your CodePush deployment key for Android (hit <ENTER> to ignore)
```

如果忘记的话，可以通过如下命令查看 CodePush 应用的 deployment key，效果如图 12.9 所示。

~ code-push deployment ls CodePush-iOS -k

| Name | Deployment Key | Update Metadata | Install Metrics |
|------------|---------------------------------------|---------------------|----------------------|
| Production | WOXLKgBm0VP22joECJQVowPFiJd4krrevS6b | No updates released | No installs recorded |
| Staging | 0RenElCPn514UTFggulH4z1ps6Qm4krrevS6b | No updates released | No installs recorded |

~ code-push deployment ls CodePush-Android -k

| Name | Deployment Key | Update Metadata | Install Metrics |
|------------|---------------------------------------|---------------------|----------------------|
| Production | gz16pfadUMLdMd-YYjcRmIXg8uEW4krrevS6b | No updates released | No installs recorded |
| Staging | ku0J5ndA3s_07hHuW0WeG60FV5Jl4krrevS6b | No updates released | No installs recorded |

图 12.9 查看 CodePush 应用的 deployment key

成功添加 react-native-code-push 到 CodePush 项目后，还需要对原生工程做一些修改，12.4.4 节将接着介绍。

12.4.4 更改 iOS 应用

首先，使用 Xcode 打开 ios/CodePush.xcodeproj 文件。打开 AppDelegate 文件，可以看到此时 jsCodeLocation 相关代码如下：

```
01 #ifdef DEBUG
02     jsCodeLocation = [[RCTBundleURLProvider sharedSettings] jsBundle
```

```

        URLForBundleRoot:@"index.ios" fallbackResource:nil];
03 #else
04     jsCodeLocation = [CodePush bundleURL];
05 #endif

```

此时在非 Debug 情况下，加载的 jsCodeLocation 是[CodePush bundleURL]。于是为了加载 CodePush 的 Bundle 地址，修改编译选项为 Release，打开 Xcode 的菜单 CodePush→EditScheme，效果如图 12.10 所示。

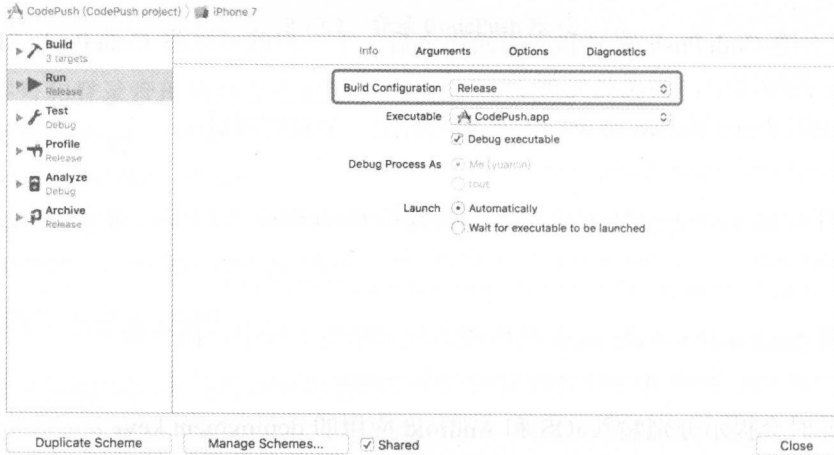


图 12.10 修改编译选项

完成了原生项目的配置之后，再修改 React Native 相关的逻辑，修改 index.ios.js 代码如下：

```

01 import React, {Component} from 'react';
02 import {AppRegistry, StyleSheet, Text, View} from 'react-native';
03 import codePush from 'react-native-code-push';
04
05 export default class CodePush extends Component {
06     constructor(props) {
07         super(props);
08         this.state = {
09             message: ''
10         }
11     }
12
13     componentDidMount() {
14         codePush.checkForUpdate().then((update) => {
15             if (!update) {
16                 this.setState({message: '已经是最新版'});
17             } else {
18                 this.setState({message: '有更新'});
19             }
20         });
21     }
22
23     render() {
24         return (
25             <View style={styles.container}>
26                 <Text style={styles.welcome}>
27                     版本号 1.0

```

```

28         </Text>
29         <Text style={styles.instructions}>
30             {this.state.message}
31         </Text>
32     </View>
33     );
34 }
35 }
36
37 const styles = StyleSheet.create({
38     container: {
39         flex: 1,
40         justifyContent: 'center',
41         alignItems: 'center',
42         backgroundColor: '#F5FCFF'
43     },
44     welcome: {
45         fontSize: 30,
46         textAlign: 'center',
47         margin: 10
48     },
49     instructions: {
50         fontSize: 30,
51         textAlign: 'center',
52         color: '#333333',
53         marginBottom: 5
54     }
55 });
56
57 AppRegistry.registerComponent('CodePush', () => codePush(CodePush));

```

上述代码将在 `componentDidMount` 中检查当前 `CodePush` 应用是否有更新，并且使用 `codePush()` 方法返回一个封装后的组件 `codePush(CodePush)`，该组件可以自动检测并下载 `CodePush` 应用的更新。

重新编译和运行应用，效果如图 12.11 所示。

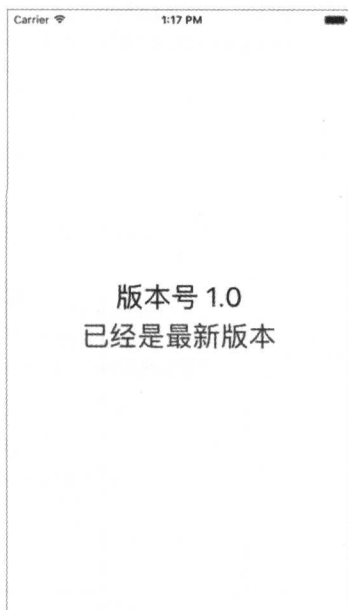


图 12.11 版本 1.0 的 iOS 应用

然后将显示的系统版本号升级为 1.1，修改 index.ios.js 代码如下：

```
01 // 这里省略了没有修改的代码
02
03 export default class CodePush extends Component {
04   // 这里省略了没有修改的代码
05
06   render() {
07     return (
08       <View style={styles.container}>
09         <Text style={styles.welcome}>
10           版本号 1.1
11         </Text>
12         <Text style={styles.instructions}>
13           {this.state.message}
14         </Text>
15       </View>
16     );
17   }
18 }
19
20 //这里省略了没有修改的代码
```

接着使用 CodePush 命令行工具发布该更新：

```
code-push release-react CodePush-iOS ios
```

此时关闭并重新打开应用，可以看到应用已检测到更新，效果如图 12.12 所示。

在检测到更新的同时，CodePush SDK 会自动下载该更新，因此当再次关闭并重新打开应用时，可以看到应用已经更新成功，效果如图 12.13 所示。



图 12.12 检测到应用更新



图 12.13 下载并使用更新

12.4.5 更改 Android 应用

和 iOS 工程相同，Android 工程的相关配置，在使用 `react-native link` 命令时也已经配置好了，所以直接修改 React Native 相关的逻辑即可，修改 `index.android.js` 代码如下：

```

01 import React, {Component} from 'react';
02 import {AppRegistry, StyleSheet, Text, View} from 'react-native';
03 import codePush from 'react-native-code-push';
04
05 export default class CodePush extends Component {
06   constructor(props) {
07     super(props);
08     this.state = {
09       message: ''
10     }
11   }
12
13   componentDidMount() {
14     codePush.checkForUpdate().then((update) => {
15       if (!update) {
16         this.setState({message: '已经是最新版'});
17       } else {
18         this.setState({message: '有更新'});
19       }
20     });
21   }
22
23   render() {
24     return (
25       <View style={styles.container}>
26         <Text style={styles.welcome}>
27           版本号 1.0
28         </Text>
29         <Text style={styles.instructions}>
30           {this.state.message}
31         </Text>
32       </View>
33     );
34   }
35 }
36
37 const styles = StyleSheet.create({
38   container: {
39     flex: 1,
40     justifyContent: 'center',
41     alignItems: 'center',
42     backgroundColor: '#F5FCFF'
43   },
44   welcome: {
45     fontSize: 30,
46     textAlign: 'center',
47     margin: 10
48   },
49   instructions: {
50     fontSize: 30,
51     textAlign: 'center',
52     color: '#333333',
53     marginBottom: 5

```

```

54     }
55   });
56
57   AppRegistry.registerComponent('CodePush', () => codePush(CodePush));

```

重新编译和运行应用，效果如图 12.14 所示。

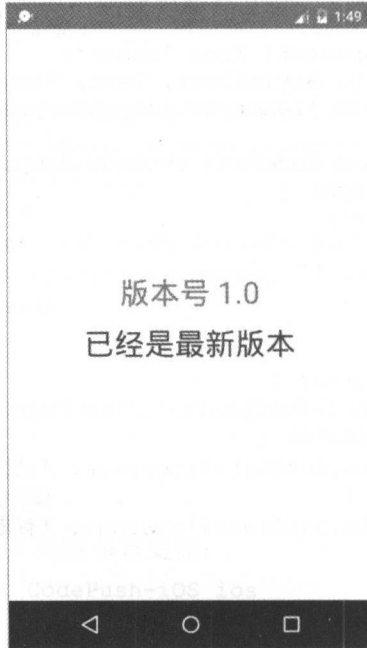


图 12.14 版本 1.0 的 Android 应用

然后将显示的系统版本号升级为 1.1，修改 `index.android.js` 代码如下：

```

01 // 这里省略了没有修改的代码
02
03 export default class CodePush extends Component {
04   // 这里省略了没有修改的代码
05
06   render() {
07     return (
08       <View style={styles.container}>
09         <Text style={styles.welcome}>
10           版本号 1.1
11         </Text>
12         <Text style={styles.instructions}>
13           {this.state.message}
14         </Text>
15       </View>
16     );
17   }
18 }
19
20 //这里省略了没有修改的代码

```

接着使用 `CodePush` 命令行工具发布该更新：

```
code-push release-react CodePush-Android android
```

此时关闭并重新打开应用，可以看到应用已检测到更新，效果如图 12.15 所示。

在检测到更新的同时，CodePush SDK 会自动下载该更新，因此当再次关闭并重新打开应用时，可以看到应用已经更新成功，效果如图 12.16 所示。

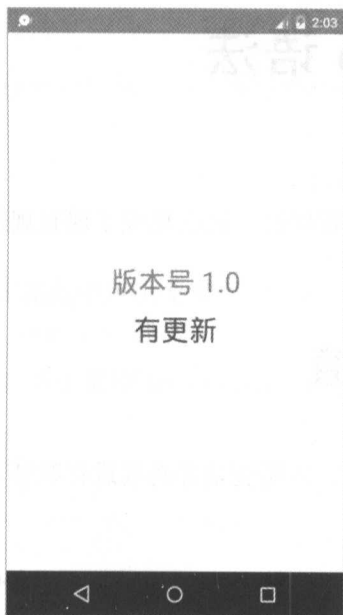


图 12.15 检测到应用更新

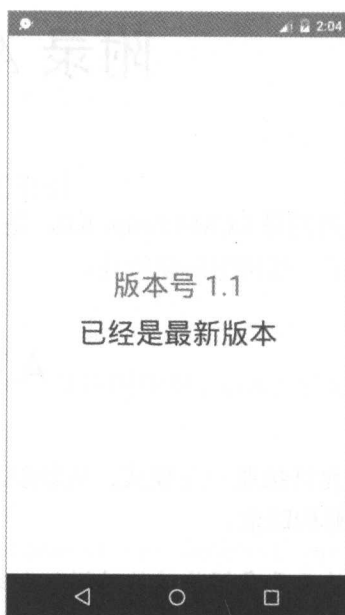



图 12.16 下载并使用更新

至此，基于 CodePush 的 React Native 应用热更新功能就完成了。

 **提示：**关于 CodePush 更多功能的使用和介绍，读者可以参考官方文档 <http://microsoft.github.io/code-push/docs/getting-started.html>。

12.5 小 结

通过本章的介绍，想必读者对 React Native 有了一个新的认识：不仅开发效率高，发布效率也很高。而且从技术实现角度来看，React Native 应用实现热更新的成本远低于原生应用，因此 React Native 作为一个新兴的移动开发平台，得到了很多开发者和项目管理者的青睐。

至此，本书也将告一段落，但是希望读者和 React Native 的缘分仍然能继续延续下去。

附录 A ES 6 语法

ES 6 指的是 ECMAScript 6.0，是 JavaScript 的最新标准，它在集成了现有规范的前提下还引入了一些实用的新特性。

A.1 解构赋值

ES 6 允许按照一定模式，从数组和对象中提取值、声明变量并对其进行赋值，这种语法称之为解构赋值。

提示：关于更多解构赋值的详细介绍，可以参考 https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment。

ES 5 规范中导入组件的代码片段如下：

```
01 var React = require('react-native');
02 var View = React.View;
```

使用解构赋值可以让代码更加优雅，效果如下：

```
01 var {View} = require('react-native');
```

A.2 导入模块

ES 5 规范中导入模块使用 `require`，例如：

```
01 var otherComponent = require('./other_component');
```

ES 6 语法采用 `import` 来代替 `require`。

```
01 import otherComponent from './other_component';
```

A.3 导出模块

ES 5 规范中导出模块使用 `module.exports`，例如：

```
01 var myComponent = React.createClass({
02   ...
03 });
04
05 module.exports = MyComponent;
```


ES 6 语法采用 `export` 来代替 `module.exports`。

```
01 var myComponent = React.createClass({
02   ...
03 });
04
05 export default MyComponent;
```

A.4 let 和 const

ES 5 规范中声明变量使用 `var`，例如

```
01 var a = 1;
```

ES 6 语法采用 `let` 和 `const` 来代替 `var`，其中 `let` 用来声明变量，`const` 用来声明只读常量。

```
01 let a = 1;
02 a = 2;
03 const PI = 3.1415;
04 PI = 3; // TypeError: Assignment to constant variable.
```

A.5 函数简写

ES 5 规范中声明函数的方法如下：

```
01 render: function() {
02   return <Text>Hello React Native!</Text>
03 }
```

ES 6 语法可以使用函数简写的声明方式。

```
01 render() {
02   return <Text>Hello React Native!</Text>
03 }
```

A.6 箭头函数

ES 5 规范中为了使函数的上下文保持一致，通常需要使用 `bind()` 函数。例如回调函数：

```
01 let callback = function(v) {
02   console.log('Hello React Native');
03 }.bind(this);
```

ES 6 语法可以使用箭头函数实现上下文的自动绑定。

```
01 let callback = () => {
02   console.log('Hello React Native');
03 };
```

A.7 字符串插值

ES 5 规范中使用如下代码拼接字符串：

```
01 let s1 = 'React Native';
02 let s2 = 'Hello ' + s1; // s2 = 'Hello React Native'
```

ES 6 语法中可以使用字符串插值的方法，通过反引号修饰字符串，然后就可以使用 `${}` 语法插入变量了。

```
01 let s1 = 'React Native';
02 let s2 = `Hello ${s1}`; // s2 = 'Hello React Native'
```

A.8 Promise 异步

Promise 是异步编程的一种解决方案，比传统的解决方案（回调函数和事件）更合理、更强大。ES 6 将其写进了语言标准，统一了用法，原生提供了 **Promise** 对象。


在使用 **Promise** 之前，异步操作的代码看起来是这样的：

```
01 try {
02     this.doAsyncOperation(params, this.onSuccessCallback, this.onError
    Callback);
03 } catch (e) {
04     ...
05 }
06
07 onSuccessCallback(params) {
08     ...
09 }
10
11 onErrorCallback (params) {
12     ...
13 }
```

如果嵌套多层异步操作的话，那么就有可能产生“回调金字塔”，即 **Callback Hell** (<http://callbackhell.com/>) 的问题。

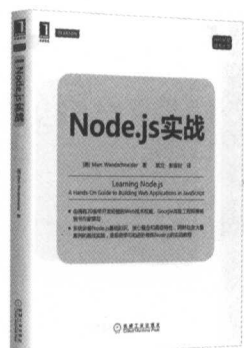
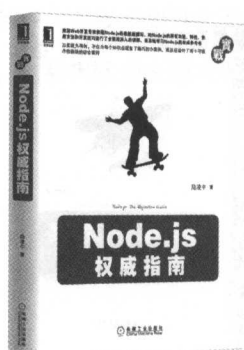
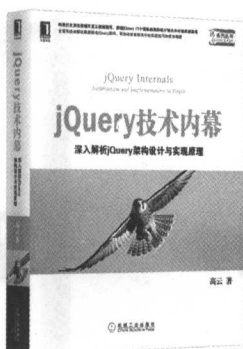
ES 6 语法中可以使用 **Promise** 来解决异步代码的问题：

```
01 this.doAsyncOperation(params).then((params) => {
02     ...
03 }).catch((e) => {
04     ...
05 })
```

 提示：关于更多 ES 6 语法的介绍，读者可以参考 ES 6 官方文档 (<http://ES 6-features.org/>) 或开源教程《ECMAScript 6 入门》 (<http://ES 6.ruanyifeng.com/>)。

Web前端开发&设计经典

框架篇



Web前端开发&设计经典

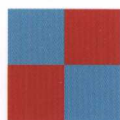
HTML5&CSS3篇



作者简介

袁林

2010年毕业于南京邮电大学。毕业后一直从事移动App研发工作，先后服务于中兴通讯、三星电子和南京企友等公司。历任App高级工程师、项目经理等职位。具备丰富的Node.js后端服务构建、Native客户端开发和React Native客户端开发经验。长期致力于应用各种IT新技术提升生产效率和解决实际问题。曾经带队自主研发多个电信级企业应用。



React Native

移动开发实战

—— 本书精华内容 ——

- React Native的优势
- 搭建React Native开发环境
- React Native开发基础知识
- React Native的组件
- 原生平台的适配和调试
- React Native的服务器端处理
- 常用React Native API
- React Native与原生平台混合编程
- 电商App的复盘
- App的发布
- App的热部署
- ES 6语法



CS



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/移动开发

ISBN 978-7-111-57179-7



9 787111 571797 >

定价: 69.00元